DSP System Toolbox™

Reference

R2012b

MATLAB®
&SIMULINK®

MathWorks®

How to Contact MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

DSP System Toolbox™ Reference

© COPYRIGHT 2011–2012 by MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

April 2011	Online only	Revised for Version 8.0 (R2011a)
September 2011	Online only	Revised for Version 8.1 (R2011b)
March 2012	Online only	Revised for Version 8.2 (R2012a)
September 2012	Online only	Revised for Version 8.3 (R2012b)

Contents

Blocks — Alphabetical List

2

System object Reference

3

Analysis Methods for Filter System Objects

4

Alphabetical List

5

Function Reference

6

Functions — Alphabetical List

7

Reference for the Properties of Filter Objects

8

Glossary

Index

1

Block Reference

Estimation

Linear Prediction

Autocorrelation LPC   Determine coefficients of Nth-order
                                              forward linear predictors

Levinson-Durbin   Solve linear system of equations
                                              using Levinson-Durbin recursion

LPC to LSF/LSP Conversion   Convert linear prediction coefficients
                                              to line spectral pairs or line spectral
                                              frequencies

LPC to/from Cepstral Coefficients   Convert linear prediction coefficients
                                              to cepstral coefficients or cepstral
                                              coefficients to linear prediction
                                              coefficients

LPC to/from RC   Convert linear prediction coefficients
                                              to reflection coefficients or reflection
                                              coefficients to linear prediction
                                              coefficients

LPC/RC to Autocorrelation   Convert linear prediction coefficients
                                              or reflection coefficients to
                                              autocorrelation coefficients

LSF/LSP to LPC Conversion   Convert line spectral frequencies
                                              or line spectral pairs to linear
                                              prediction coefficients

Parametric Estimation

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

Power Spectrum Estimation

Burg Method	Power spectral density estimate using Burg method
Covariance Method	Power spectral density estimate using covariance method
Magnitude FFT	Compute nonparametric estimate of spectrum using periodogram method
Modified Covariance Method	Power spectral density estimate using modified covariance method
Periodogram	Power spectral density or mean-square spectrum estimate using periodogram method
Yule-Walker Method	Power spectral density estimate using Yule-Walker method

Filtering

Adaptive Filters

Block LMS Filter   Compute output, error, and weights
                            using LMS adaptive algorithm

Fast Block LMS Filter   Compute output, error, and weights
                            using LMS adaptive algorithm

Kalman Filter   Predict or estimate states of dynamic
                            systems

LMS Filter   Compute output, error, and weights
                            using LMS adaptive algorithm

RLS Filter   Compute filtered output, filter error,
                            and filter weights for given input and
                            desired signal using RLS adaptive
                            filter algorithm

Filter Designs

Arbitrary Response Filter   Design arbitrary response filter

Audio Weighting Filter   Design audio weighting filter

Bandpass Filter   Design bandpass filter

Bandstop Filter   Design bandstop filter

CIC Compensator   Design CIC compensator

CIC Filter	Design Cascaded Integrator-Comb (CIC) Filter
Comb Filter	Design comb Filter
Differentiator Filter	Design differentiator filter
Halfband Filter	Design halfband filter
Highpass Filter	Design highpass filter
Hilbert Filter	Design Hilbert filter
Inverse Sinc Filter	Design inverse sinc filter
Lowpass Filter	Design lowpass filter
Nyquist Filter	Design Nyquist filter
Octave Filter	Design octave filter
Parametric Equalizer	Design parametric equalizer
Peak-Notch Filter	Design peak or notch filter
Pulse Shaping Filter	Design pulse shaping filter

Filter Implementations

Allpole Filter	Model allpole filters
Analog Filter Design	Design and implement analog filters
Biquad Filter	Model biquadratic IIR (SOS) filters
Digital Filter	Filter each channel of input over time using static or time-varying digital filter implementations
Digital Filter Design	Design and implement digital FIR and IIR filters
Discrete FIR Filter	Model FIR filters
Filter Realization Wizard	Construct filter realizations using Digital Filter block or Sum, Gain, and Delay blocks

Overlap-Add FFT Filter	Implement overlap-add method of frequency-domain filtering
Overlap-Save FFT Filter	Implement overlap-save method of frequency-domain filtering

Multirate Filters

CIC Decimation	Decimate signal using Cascaded Integrator-Comb filter
CIC Interpolation	Interpolate signal using Cascaded Integrator-Comb filter
Dyadic Analysis Filter Bank	Decompose signals into subbands with smaller bandwidths and slower sample rates or compute discrete wavelet transform (DWT)
Dyadic Synthesis Filter Bank	Reconstruct signals from subbands with smaller bandwidths and slower sample rates or compute inverse discrete wavelet transform (IDWT)
FIR Decimation	Filter and downsample input signals
FIR Interpolation	Upsample and filter input signals
FIR Rate Conversion	Upsample, filter, and downsample input signals
Two-Channel Analysis Subband Filter	Decompose signal into high-frequency and low-frequency subbands
Two-Channel Synthesis Subband Filter	Reconstruct signal from high-frequency and low-frequency subbands

Math Functions

Math Operations

Complex Exponential	Compute complex exponential function
Cumulative Product	Cumulative product of channel, column, or row elements
Cumulative Sum	Cumulative sum of channel, column, or row elements
dB Conversion	Convert magnitude data to decibels (dB or dBm)
dB Gain	Apply decibel gain
Difference	Compute element-to-element difference along specified dimension of input
Normalization	Perform vector normalization along rows, columns, or specified dimension

Matrices and Linear Algebra

Linear System Solvers

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

Matrix Factorizations

Cholesky Factorization	Factor square Hermitian positive definite matrix into triangular components
LDL Factorization	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
LU Factorization	Factor square matrix into lower and upper triangular components
QR Factorization	Factor arbitrary matrix into unitary and upper triangular components
Singular Value Decomposition	Factor matrix using singular value decomposition

Matrix Inverses

Cholesky Inverse	Compute inverse of Hermitian positive definite matrix using Cholesky factorization
LDL Inverse	Compute inverse of Hermitian positive definite matrix using LDL factorization
LU Inverse	Compute inverse of square matrix using LU factorization
Pseudoinverse	Compute Moore-Penrose pseudoinverse of matrix

Matrix Operations

Array-Vector Add	Add vector to array along specified dimension
Array-Vector Divide	Divide array by vector along specified dimension
Array-Vector Multiply	Multiply array by vector along specified dimension
Array-Vector Subtract	Subtract vector from array along specified dimension
Constant Diagonal Matrix	Generate square, diagonal matrix
Create Diagonal Matrix	Create square diagonal matrix from diagonal elements
Extract Diagonal	Extract main diagonal of input matrix
Extract Triangular Matrix	Extract lower or upper triangle from input matrices
Identity Matrix	Generate matrix with ones on main diagonal and zeros elsewhere
Matrix 1-Norm	Compute 1-norm of matrix

Matrix Concatenate	Concatenate input signals of same data type to create contiguous output signal
Matrix Exponential	Compute matrix exponential
Matrix Multiply	Multiply or divide inputs
Matrix Product	Multiply matrix elements along rows, columns, or entire input
Matrix Square	Compute square of input matrix
Matrix Sum	Sum matrix elements along rows, columns, or entire input
Matrix Sum (Obsolete)	Sum matrix elements along rows, columns, or entire input
Overwrite Values	Overwrite submatrix or subdiagonal of input
Permute Matrix	Reorder matrix rows or columns
Reciprocal Condition	Compute reciprocal condition of square matrix in 1-norm
Submatrix	Select subset of elements (submatrix) from matrix input
Toeplitz	Generate matrix with Toeplitz symmetry
Transpose	Transpose matrix

Polynomial Functions

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

Quantizers

G711 Codec	Quantize narrowband speech input signals
Quantizer	Discretize input at specified interval
Scalar Quantizer Decoder	Convert each index value into quantized output value
Scalar Quantizer Design	Start Scalar Quantizer Design Tool (SQDTool) to design scalar quantizer using Lloyd algorithm
Scalar Quantizer Encoder	Encode each input value by associating it with index value of quantization region
Uniform Decoder	Decode integer input into floating-point output
Uniform Encoder	Quantize and encode floating-point input into integer output
Vector Quantizer Decoder	Find vector quantizer codeword that corresponds to given, zero-based index value
Vector Quantizer Design	Design vector quantizer using Vector Quantizer Design Tool (VQDTool)
Vector Quantizer Encoder	For given input, find index of nearest codeword based on Euclidean or weighted Euclidean distance measure

Signal Management

Buffers

Buffer   Buffer input sequence to smaller or
                                      larger frame size

Delay Line   Rebuffer sequence of inputs

Queue   Store inputs in FIFO register

Stack   Store inputs into LIFO register

Unbuffer   Unbuffer input frame into sequence
                                      of scalar outputs

Indexing

Flip   Flip input vertically or horizontally

Multiport Selector   Distribute arbitrary subsets of input
                                      rows or columns to multiple output
                                      ports

Overwrite Values   Overwrite submatrix or subdiagonal
                                      of input

Selector   Select input elements from vector,
                                      matrix, or multidimensional signal

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

Signal Attributes

Check Signal Attributes	Error when input signal does or does not match selected attributes exactly
Convert 1-D to 2-D	Reshape 1-D or 2-D input to 2-D matrix with specified dimensions
Convert 2-D to 1-D	Convert 2-D matrix input to 1-D vector
Data Type Conversion	Convert input signal to specified data type
Frame Conversion	Specify sampling mode of output signal
Inherit Complexity	Change complexity of input to match reference signal

Switches and Counters

Counter	Count up or down through specified range of numbers
Edge Detector	Detect transition from zero to nonzero value
Event-Count Comparator	Detect threshold crossing of accumulated nonzero inputs
Multiphase Clock	Generate multiple binary clock signals

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

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

Signal Operations

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

Variable Integer Delay	Delay input by time-varying integer number of sample periods
Window Function	Compute and/or apply window to input signal
Zero Crossing	Count number of times signal crosses zero in single time step

Sinks

Display	Show value of input
Matrix Viewer	Display matrices as color images
Signal To Workspace	Write data to MATLAB workspace
Spectrum Scope	Display mean-square spectrum or power spectral density of each input signal
Time Scope	Display time-domain signals
To Audio Device	Write audio data to computer's audio device
To Multimedia File	Write video frames and audio samples to multimedia file
Triggered To Workspace	Write input sample to MATLAB workspace when triggered
UDP Send	Send UDP message
Vector Scope	Display vector or matrix of time-domain, frequency-domain, or user-defined data
Waterfall	View vectors of data over time

Sources

Chirp	Generate swept-frequency cosine (chirp) signal
Constant	Generate constant value
Constant Diagonal Matrix	Generate square, diagonal matrix
Discrete Impulse	Generate discrete impulse
DSP Constant (Obsolete)	Generate discrete- or continuous-time constant signal
From Audio Device	Read audio data from computer's audio device
From Multimedia File	Read multimedia file
Identity Matrix	Generate matrix with ones on main diagonal and zeros elsewhere
MIDI Controls	Output values from controls on MIDI control surface
Multiphase Clock	Generate multiple binary clock signals
N-Sample Enable	Output ones or zeros for specified number of sample times
Random Source	Generate randomly distributed values
Signal From Workspace	Import signal from MATLAB workspace
Sine Wave	Generate continuous or discrete sine wave
UDP Receive	Receive `uint8` vector as UDP message

Statistics

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

Transforms

|	
Analytic Signal	Compute analytic signals of discrete-time inputs
Complex Cepstrum	Compute complex cepstrum of input
DCT	Discrete cosine transform (DCT) of input
DWT	Discrete wavelet transform (DWT) of input or decompose signals into subbands with smaller bandwidths and slower sample rates
FFT	Fast Fourier transform (FFT) of input
IDCT	Inverse discrete cosine transform (IDCT) of input
IDWT	Inverse discrete wavelet transform (IDWT) of input or reconstruct signals from subbands with smaller bandwidths and slower sample rates
IFFT	Inverse fast Fourier transform (IFFT) of input
Inverse Short-Time FFT	Recover time-domain signals by performing inverse short-time, fast Fourier transform (FFT)
Magnitude FFT	Compute nonparametric estimate of spectrum using periodogram method
Real Cepstrum	Compute real cepstrum of input
Short-Time FFT	Nonparametric estimate of spectrum using short-time, fast Fourier transform (FFT) method

Blocks — Alphabetical List

Allpole Filter

Purpose	Model allpole filters
Library	Filtering / Filter Implementations
	dsparch4

Description
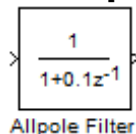
The Allpole Filter block independently filters each channel of the input signal with the specified allpole filter. The block can implement static filters with fixed coefficients, as well as time-varying filters with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. The **Input processing** parameter allows you to specify whether the block treats each element of the input as an independent channel (sample-based processing), or each column of the input as an independent channel (frame-based processing).

The outputs of this block numerically match the outputs of the DSP System Toolbox™ Digital Filter block.

This block supports the Simulink® state logging feature. See "States" in the *Simulink User's Guide* for more information.

Filter Structure Support

You can change the filter structure implemented with the Allpole Filter block by selecting one of the following from the **Filter structure** parameter:

- Direct form
- Direct form transposed
- Lattice AR

Specifying Initial States

The Allpole Filter block initializes the internal filter states to zero by default, which has the same effect as assuming that past inputs and outputs are zero. You can optionally use the **Initial states** parameter to specify nonzero initial conditions for the filter delays.

To determine the number of initial states you must specify and how to specify them, see the table on valid initial states. The **Initial states** parameter can take one of the forms described in the next table.

Valid Initial States

Initial Condition	Description
Scalar	The block initializes all delay elements in the filter to the scalar value.
Vector or matrix (for applying different delay elements to each channel)	Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel: • The vector length equals the product of the number of input channels and the number of delay elements in the filter, `#_of_filter_coeffs-1` (or `#_of_reflection_coeffs` for `Lattice AR`). • The matrix must have the same number of rows as the number of delay elements in the filter, `#_of_filter_coeffs-1` (`#_of_reflection_coeffs` for `Lattice AR`), and must have one column for each channel of the input signal.

Data Type Support

The Allpole Filter block accepts and outputs real and complex signals of any numeric data type supported by Simulink. The block supports the same types for the coefficients.

The following diagrams show the filter structure and the data types used within the Allpole Filter block for fixed-point signals.

Direct Form

You cannot specify the state data type on the block mask for this structure because the output states have the same data types as the output.
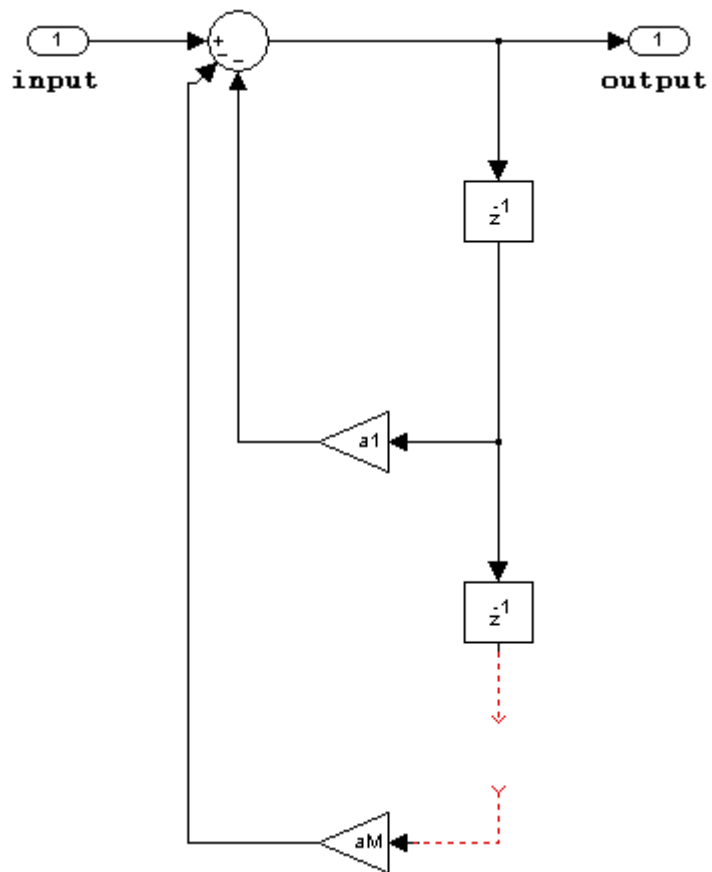
Allpole Filter
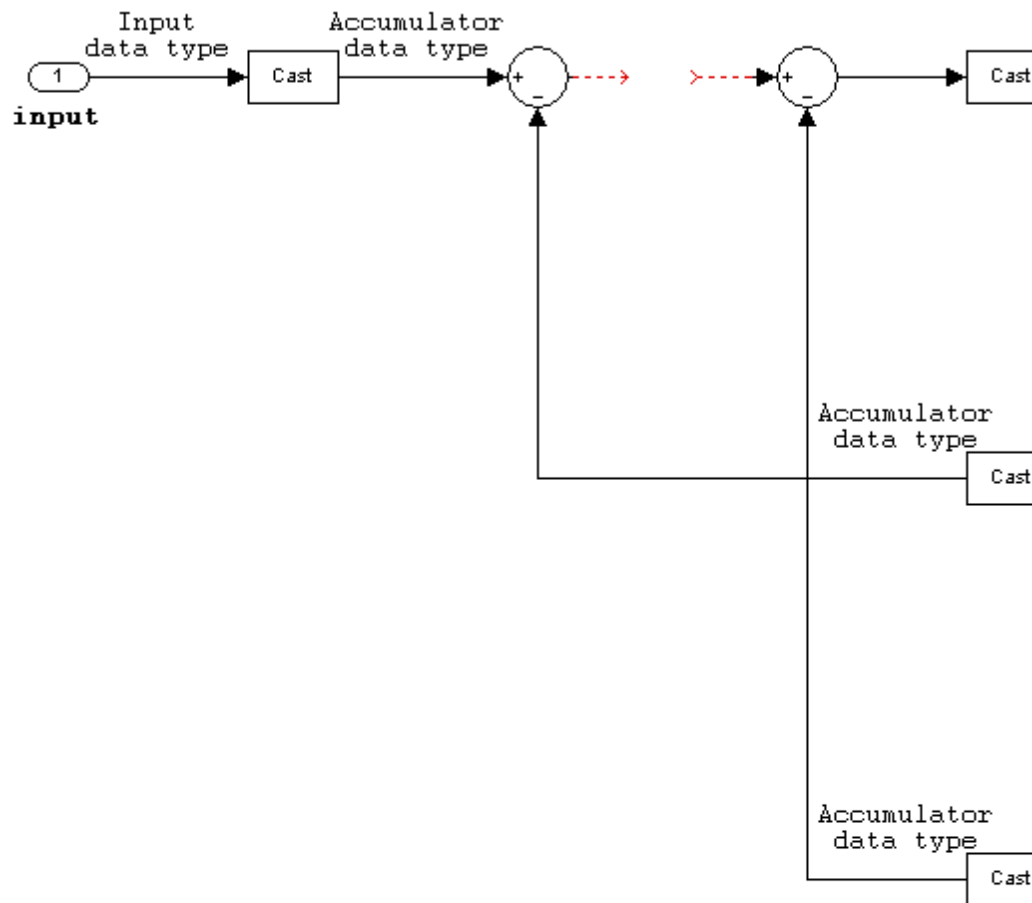
Allpole Filter

Direct Form Transposed

States are complex when either the inputs or the coefficients are complex.
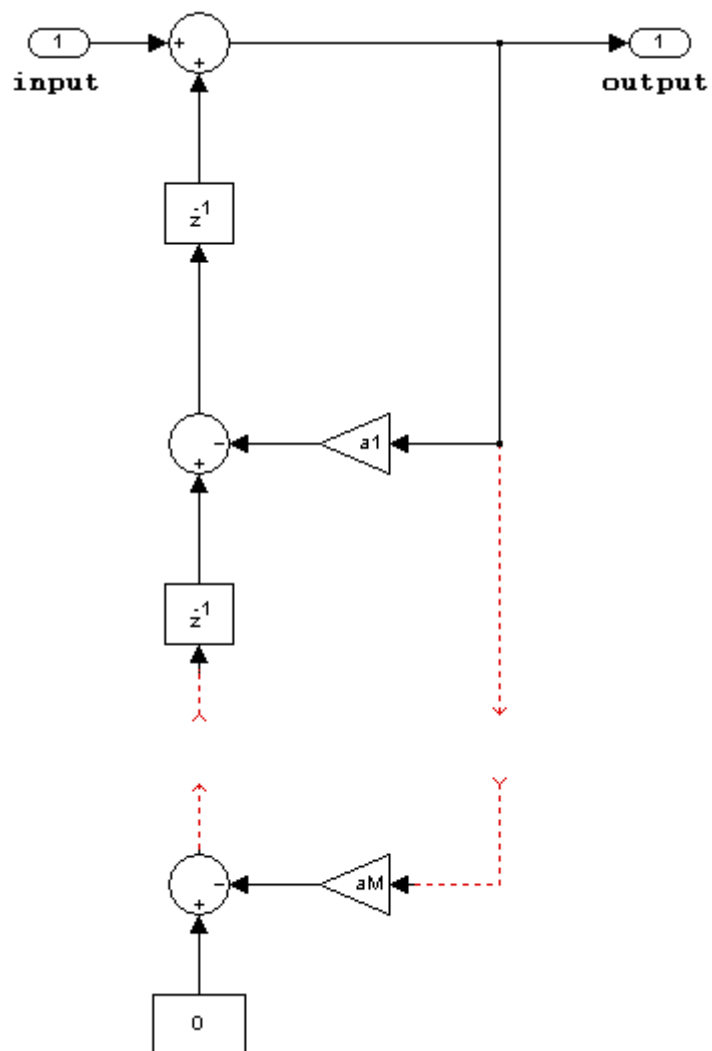
Allpole Filter
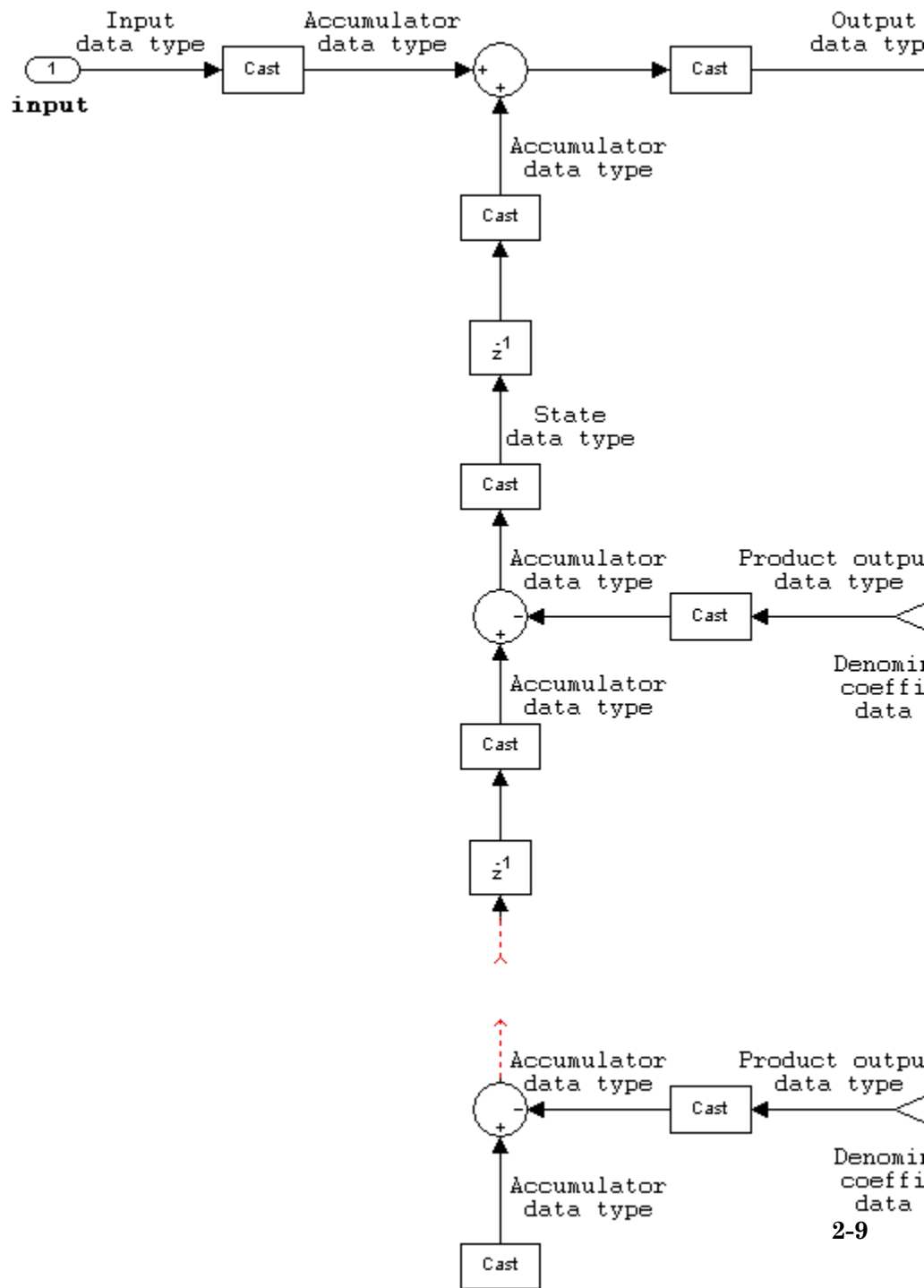
Allpole Filter

Lattice AR
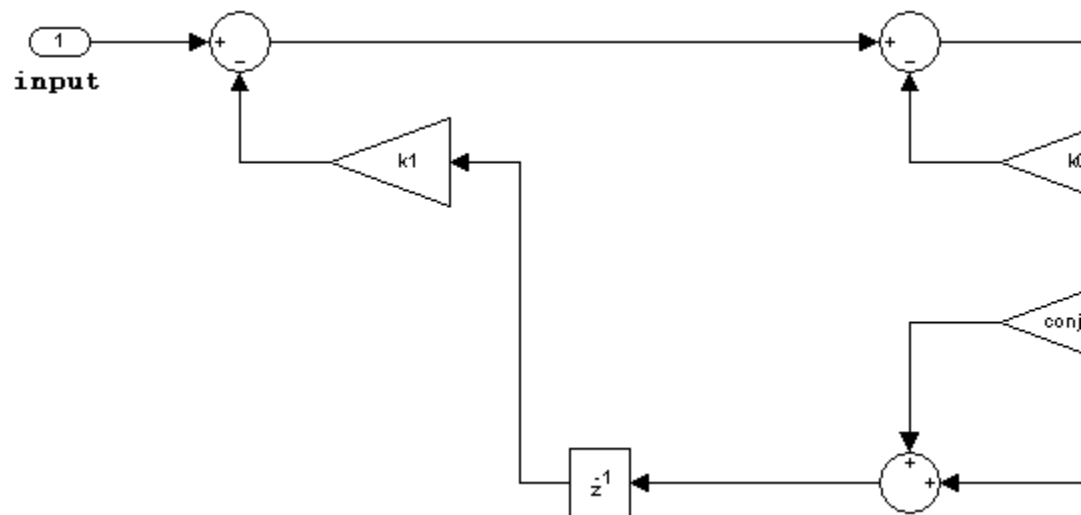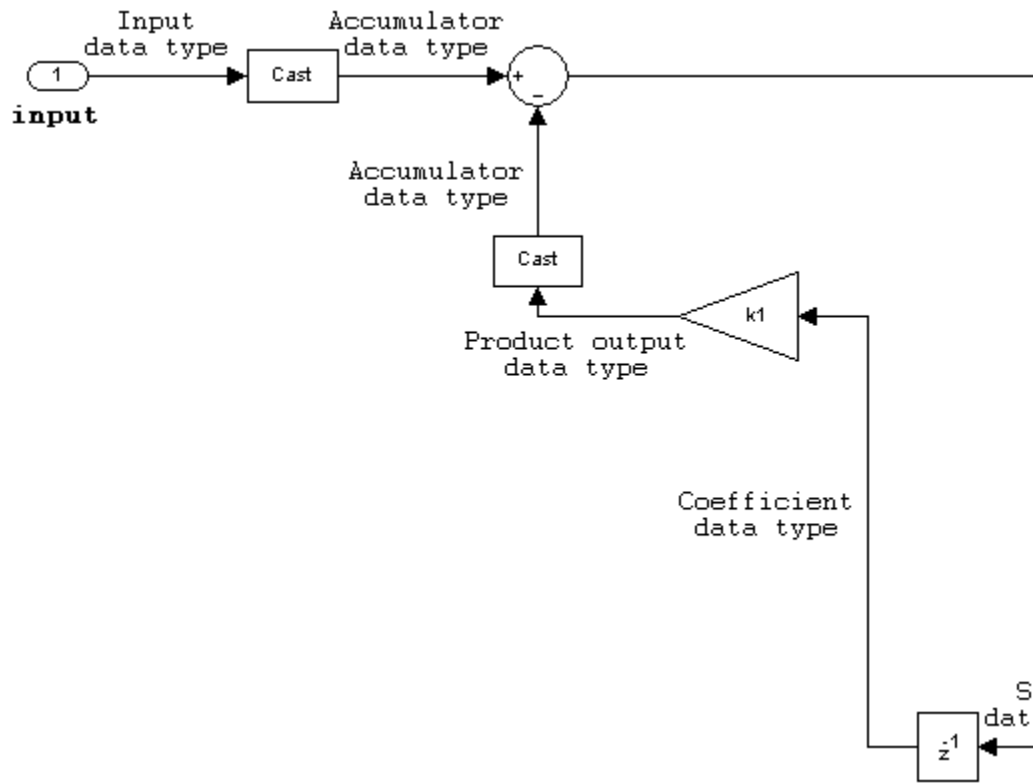
Parameters and Dialog Box

The **Main** pane of the Allpole Filter block dialog box appears as follows.

Allpole Filter
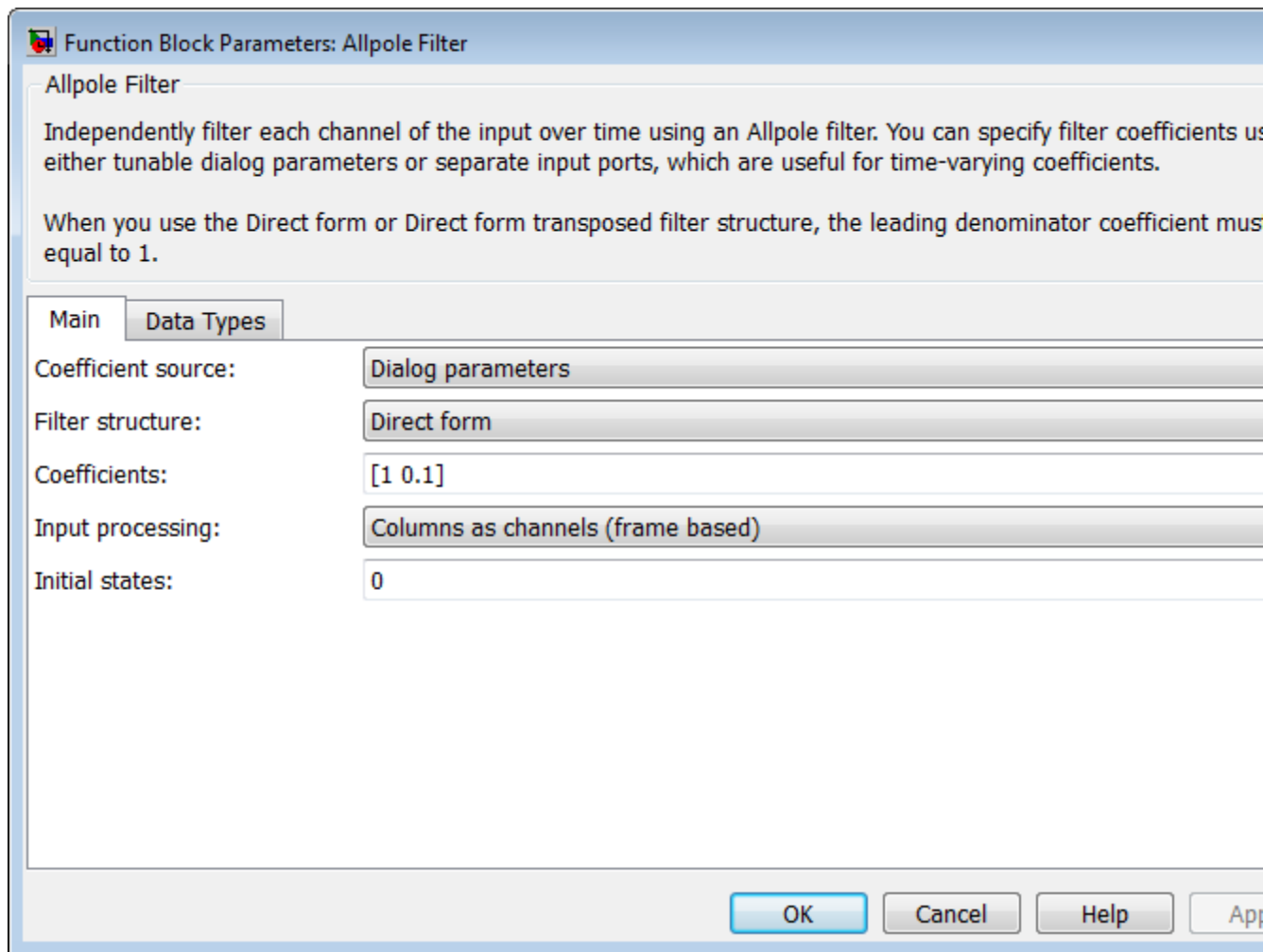
Coefficient source

Select whether you want to specify the filter coefficients on the block mask or through an input port.

Filter structure

Select the filter structure you want the block to implement. You can select `Direct form`, `Direct form transposed`, or `Lattice AR`.

Coefficients

Specify the row vector of coefficients of the filter's transfer function.

This parameter is visible only when you set the **Coefficient source** to `Dialog parameters`.

Input processing

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- `Elements as channels (sample based)` — Treat each element of the input as an independent channel (sample-based processing).

- `Columns as channels (frame based)` — Treat each column of the input as an independent channel (frame-based processing).

Initial states

Specify the initial conditions of the filter states. To learn how to specify initial states, see "Specifying Initial States" on page 2-2.

The **Data Types** pane of the Allpole Filter block dialog box appears as follows.

Allpole Filter

Independently filter each channel of the input over time using an Allpole filter. You can spec
either tunable dialog parameters or separate input ports, which are useful for time-varying

When you use the Direct form or Direct form transposed filter structure, the leading denom
equal to 1.

Main **Data Types**

Floating-point inheritance takes precedence over the settings in the "Data Type" column bel
is floating point, all block data types match the input.

	Data Type	Assistant	Minimu
Coefficients:	erit: Same word length as input ▼	>>	[]
Product output:	Inherit: Inherit via internal rule ▼	>>	
Accumulator:	Inherit: Inherit via internal rule ▼	>>	
Output:	Inherit: Same as input ▼	>>	[]

☐ Lock data type settings against changes by the fixed-point tools

Integer rounding mode: Floor ▼

☐ Saturate on integer overflow

[OK] [Cancel]

Coefficients

Specify the coefficient data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same
  word length as input

- A built-in integer, for example, `int8`

- A data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Coefficient** parameter.

See "Specify Data Types Using Data Type Assistant" for more information.

Coefficients minimum
Specify the minimum value that a filter coefficient should have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Check Parameter Values")

- Automatic scaling of fixed-point data types

Coefficients maximum
Specify the maximum value that a filter coefficient should have. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Check Parameter Values")

- Automatic scaling of fixed-point data types

Product output
Specify the product output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- A built-in data type, for example, `int8`

Allpole Filter

- A data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Product output** parameter.

See "Specify Data Types Using Data Type Assistant" for more information.

Accumulator

Specify the accumulator data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- A built-in data type, for example, `int8`

- A data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator** parameter.

See "Specify Data Types Using Data Type Assistant" for more information.

State

Specify the state data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit:  Same as accumulator`

- A built-in integer, for example, `int8`

- A data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

This parameter is only visible when the selected filter structure is `Lattice MA`.

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **State** parameter.

See "Specify Data Types Using Data Type Assistant" for more information.

Output
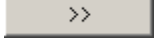
Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit:  Same as accumulator`

- A built-in data type, for example, `int8`

- A data type object, for example, a `Simulink.NumericType` object

- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

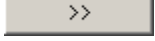Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output** parameter.

See "Specify Block Output Data Types" in the *Simulink User's Guide* for more information.

Output minimum

Specify the minimum value that the block should output. The
default value is [] (unspecified). Simulink software uses this
value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value that the block should output. The
default value is [] (unspecified). Simulink software uses this
value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from
overriding the data types you specify on the block mask.

Integer rounding mode

Specify the rounding mode for fixed-point operations. For more
information, see "Rounding" in the Simulink Fixed Point™
documentation.

Saturate on integer overflow

Action	Reasons for Taking This Action	What Happens for Overflows	Example
Select this check box.	Your model has possible overflow and you want explicit	Overflows saturate to either the minimum or maximum value	An overflow associated with a signed 8-bit

Action	Reasons for Taking This Action	What Happens for Overflows	Example
	saturation protection in the generated code.	that the data type can represent.	integer can saturate to -128 or 127.
Do not select this check box.	You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see "Checking for Signal Range Errors".	Overflows wrap to the appropriate value that is representable by the data type.	The number 130 does not fit in a signed 8-bit integer and wraps to -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit integers

See Also

Digital Filter	DSP System Toolbox
Filter Realization Wizard	DSP System Toolbox

Allpole Filter

`fdatool`	DSP System Toolbox
`fvtool`	Signal Processing Toolbox

Purpose	Design and implement analog filters
Library	Filtering / Filter Implementations
	dsparch4

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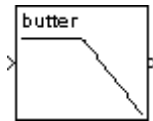
The Analog Filter Design block designs and implements a Butterworth, Chebyshev type I, Chebyshev type II, or elliptic filter in a highpass, lowpass, bandpass, or bandstop configuration.

The input must be a sample-based, continuous-time, real-valued, scalar signal.

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

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

The following table lists the available parameters for each design/band combination. For lowpass and highpass band configurations, these parameters include the passband edge frequency Ω_p, the stopband edge frequency Ω_s, the passband ripple R_p, and the stopband attenuation R_s. For bandpass and bandstop configurations, the parameters include the

Analog Filter Design

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 filter design commands in Signal Processing Toolbox™ software's `buttap`, `cheb1ap`, `cheb2ap`, and `ellipap` functions, and are implemented in state-space form. Filters of order 8 or less are implemented in controller canonical form for improved efficiency.

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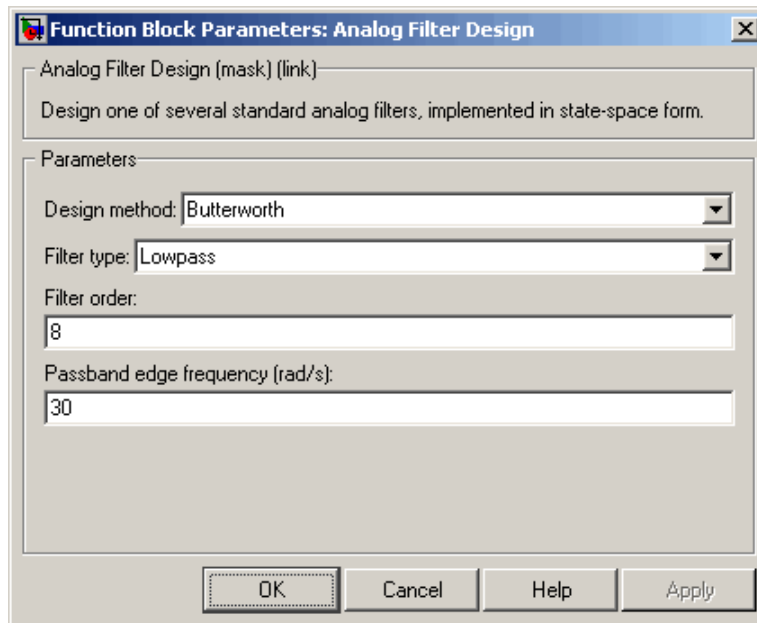
The parameters displayed in the dialog box vary for different design/band combinations. Only some of the parameters listed below are visible in the dialog box at any one time.

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

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

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

Analog Filter Design

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

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

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

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

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

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

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

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

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

Supported Data Types

- Double-precision floating point

See Also

Digital Filter Design	DSP System Toolbox
buttap	Signal Processing Toolbox
cheb1ap	Signal Processing Toolbox
cheb2ap	Signal Processing Toolbox
ellipap	Signal Processing Toolbox

See the following sections for related information:

- "Filter Design"
- "Filter Analysis"
- "Analog Filter Design Block"

Analytic Signal

Purpose   Compute analytic signals of discrete-time inputs

Library   Transforms

dspxfrm3

Description
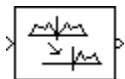
The Analytic Signal block computes the complex analytic signal corresponding to each channel of the real M-by-N input, u

$$y = u + jH\{u\}$$

where $j = \sqrt{-1}$ and $H\{\ \}$ 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 Fourier transform of the analytic signal doubles the positive frequency content of the original signal while zeroing-out negative frequencies and retaining 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 dimensions as the input.

Frame-Based Processing

When you set the **Input processing** parameter to Columns as channels (frame based), the block performs frame-based processing. In this mode, the block treats an M-by-N matrix input as N independent channels containing M sequential time samples. The block computes the analytic signal for each channel over time.

Sample-Based Processing

When you set the **Input processing** parameter to Elements as channels (sample based), the block performs sample-based processing. In this mode, the block treats an M-by-N matrix input as

M*N independent channels and computes the analytic signal for each channel (matrix element) over time.

Function Block Parameters: Analytic Signal

Analytic Signal (mask) (link)

Complex analytic signal of input.

Parameters

Filter order (must be even):

```
100
```

Input processing: Columns as channels (frame based)

| OK | Cancel | Help | Apply |
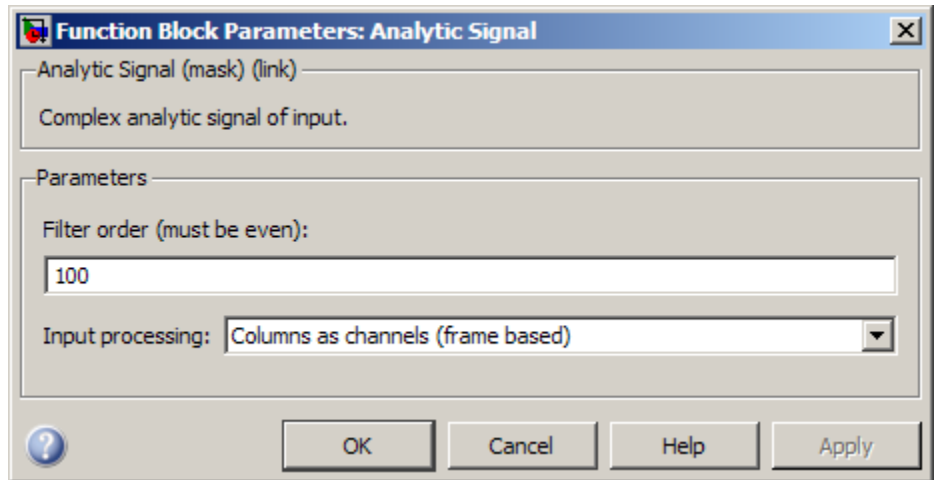
Dialog Box

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

Input processing
> Specify how the block should process the input. You can set this parameter to one of the following options:

> - Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

> - Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Analytic Signal

> **Note** The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Supported Data Types

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

Purpose　　　Design arbitrary response filter

Library　　　Filtering / Filter Designs

`dspfdesign`

Description　　This block brings the filter design capabilities of the `filterbuilder` function to the Simulink environment.
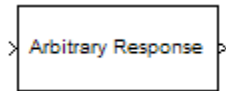
Dialog Box　　See "Arbitrary Response Filter Design Dialog Box — Main Pane" on page 7-666 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Arbitrary Response Filter

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop down list, where `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Order mode

Select `Minimum` or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to `IIR`, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

Order

Enter the order for FIR filter, or the order of the numerator for the IIR filter.

Denominator order

Select the check box and enter the denominator order. This option is enabled only if `IIR` is selected for **Impulse response**.

Arbitrary Response Filter

Filter type

This option is available for FIR filters only. Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Response Specification

Number of Bands

Select the number of bands in the filter. Multiband design is available for both FIR and IIR filters.

Specify response as

Specify the response as `Amplitudes`, `Magnitudes and phases`, `Frequency response`, or `Group delay`. `Group delay` is only available for IIR designs.

Frequency units

Specify frequency units as either `Normalized`, which means normalized by the input sampling frequency, or select from `Hz`, `kHz`, `MHz`, or `GHz`.

Input Fs

> Enter the input sampling frequency in the units specified in the **Frequency units** drop-down list. When you select the frequency units, this option is available.

Band Properties

These properties are modified automatically depending on the response chosen in the **Specify response as** drop-down list. Two or three columns are presented for input. The first column is always Frequencies. The other columns are Amplitudes, Magnitudes, Phases, or Frequency Response. Enter the corresponding vectors of values for each column.

- **Frequencies** and **Amplitudes** — These columns are presented for input if the response chosen in the **Specify response as** drop-down list is `Amplitudes`.

- **Frequencies**, **Magnitudes**, and **Phases** — These columns are presented for input if the response chosen in the **Specify response as** drop-down list is `Magnitudes and phases`.

- **Frequencies** and **Frequency response** — These columns are presented for input if the response chosen in the **Specify response as** drop-down list is `Frequency response`.

Algorithm

Design Method

> Select the design method for the filter. Different methods are enabled depending on the defining parameters entered in the previous sections.

Design Options

> The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications.

- **Window** — Replace the square brackets with the name of a window function or function handle. For example, `hamming` or `@hamming`. If the window function takes parameters other than the length, use a cell array. For example, `{`kaiser',3.5}` or `{@chebwin,60}`.

- **Density factor** — Valid when the **Design method** is `Equiripple`. Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

  Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade-off between the accurate approximation to the ideal filter and the time to design the filter.

- **Phase constraint** — Valid when the **Design method** is `Equiripple`, you have the DSP System Toolbox installed, and **Specify response as** is set to `Amplitudes`. Choose one of `Linear`, `Minimum`, or `Maximum`.

- **Weights** — Valid when the **Design method** is `Equiripple`. Uses the weights in **Weights** to weight the error for a single-band design. If you have multiple frequency bands, the **Weights** design option changes to **B1 Weights**, **B2 Weights** to designate the separate bands.

Filter Implementation

Structure
Select the structure for the filter, available for the corresponding design method.

Use basic elements to enable filter customization
Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Arbitrary Response Filter

> **Note** The Inherited (this choice will be removed — see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- Enforce single-rate processing — When you select this option, the block maintains the sample rate of the input.

- Allow multirate processing — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to Elements as channels (sample based).

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Purpose   Add vector to array along specified dimension

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3
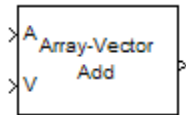
Description   The Array-Vector Add block adds the values in the specified dimension
of the N-dimensional input array *A* to the values in the input vector *V*.

The length of the input *V* must be the same as the length of the specified
dimension of *A*. The Array-Vector Add block adds each element of *V* to
the corresponding element along that dimension of *A*.

Consider a 3-dimensional *M*-by-*N*-by-*P* input array *A(i,j,k)* and a *N*-by-1
input vector *V*. When the **Add along dimension** parameter is set to 2,
the output of the block *Y(i,j,k)* is

$$Y(i, j, k) = A(i, j, k) + V(j)$$

where

$$1 \le i \le M$$
$$1 \le j \le N$$
$$1 \le k \le P$$

The output of the Array-Vector Add block is the same size as the
input array, *A*. This block accepts real and complex floating-point and
fixed-point inputs.

Fixed-Point Data Types

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

Array-Vector Add
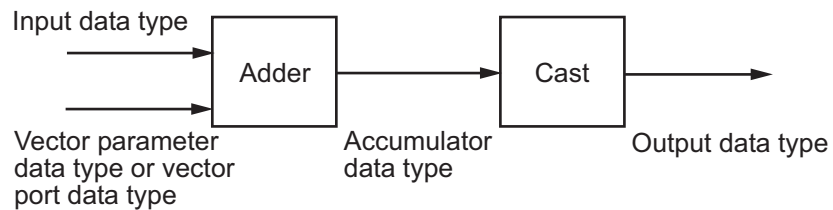
When you specify the vector *V* on the **Main** pane of the block mask, you must specify the data type and scaling properties of its elements in the **Vector (V)** parameter on the **Data Types** tab. When the vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

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

Dialog Box

The **Main** pane of the Array-Vector Add block dialog appears as follows.

Function Block Parameters: Array-Vector Add ✕

─ Array-Vector Add ──────────────────────────────────────

Add the input array A to the elements of vector V along the specified dimension. Note that unoriented input signals are treated as oriented column vectors. The output of this block is always oriented.

When you specify the vector V on the block dialog and the fixed-point data type mode is set to "Specify word length," the fixed-point scaling is automatically set for you. In these cases, the scaling is set to the best possible precision given the real-world values and word length of V. This is equivalent to the "Best Precision: Matrix-wise" scaling option used in some fixed-point Simulink blocks.

| Main | Data Types |

─ Parameters ──────────────────────────────────────

Add along dimension: | 1 |

Vector (V) source: | Input port ▾ |

OK   Cancel   Help   Apply
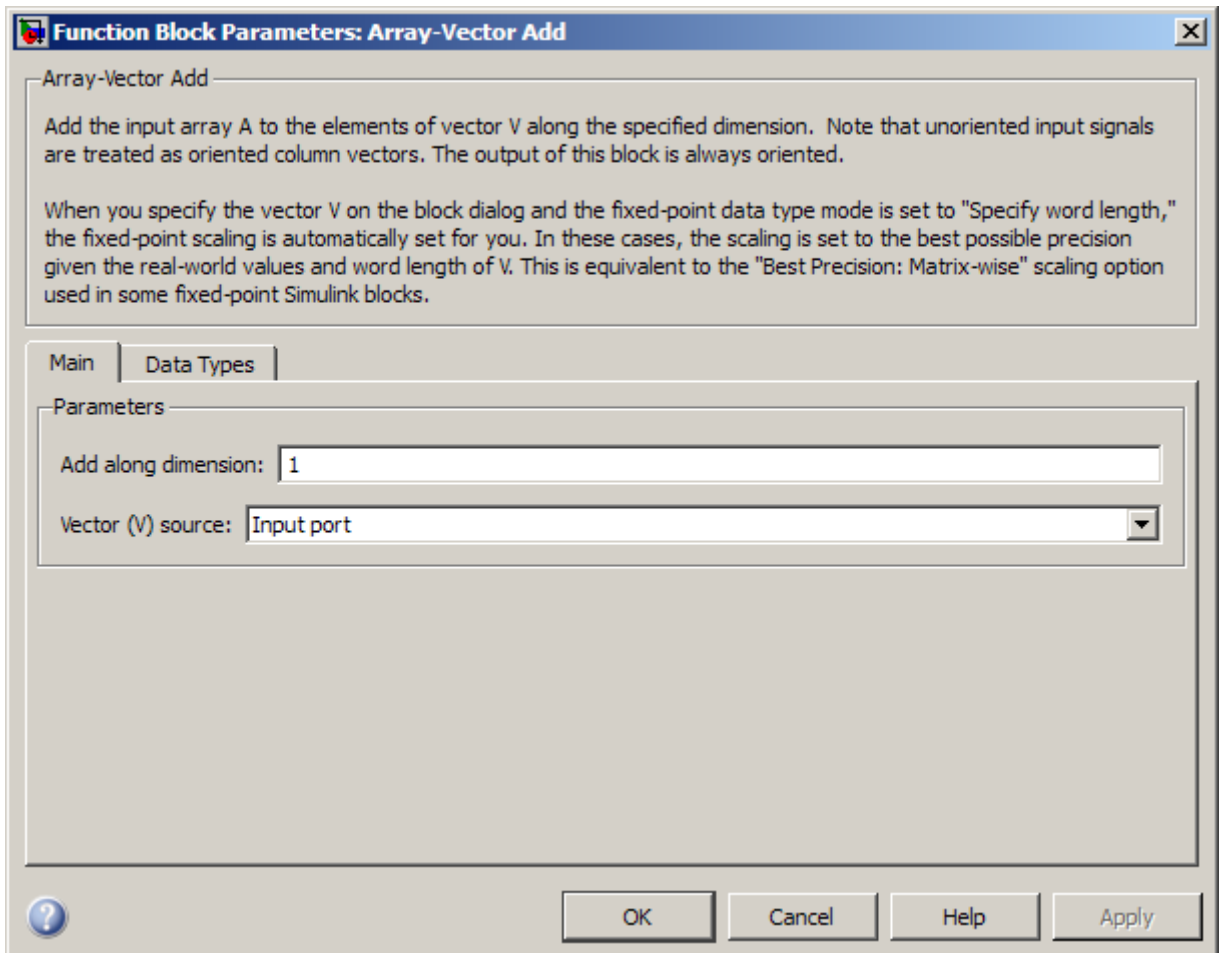
Add along dimension

Specify the dimension along which to add the input array *A* to the elements of vector *V*.

Array-Vector Add

Vector (V) source

> Specify the source of the vector, *V*. The vector can come from the `Input port` or from a `Dialog parameter`.

Vector (V)

> Specify the vector, *V*. This parameter is visible only when you select `Dialog parameter` for the **Vector (V) source** parameter.

The **Data Types** pane of the Array-Vector Add block dialog appears as follows.

Function Block Parameters: Array-Vector Add ✕

─ Array-Vector Add ─────────────────────────────────

Add the input array A to the elements of vector V along the specified dimension. Note that unoriented input signals are treated as oriented column vectors. The output of this block is always oriented.

When you specify the vector V on the block dialog and the fixed-point data type mode is set to "Specify word length," the fixed-point scaling is automatically set for you. In these cases, the scaling is set to the best possible precision given the real-world values and word length of V. This is equivalent to the "Best Precision: Matrix-wise" scaling option used in some fixed-point Simulink blocks.

Main	Data Types

─ Fixed-point operational parameters ───────────────

Rounding mode: Floor ▾   Overflow mode: Wrap ▾

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input. In addition, fixed-point accumulator attributes only apply when fixed-point block inputs are complex.

	Data Type	Assistant	Minimum	Maximum
Accumulator:	Inherit: Inherit via internal rule ▾	>>		
Output:	Inherit: Same as accumulator ▾	>>	0	0

☐ Lock data type settings against changes by the fixed-point tools

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

Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Array-Vector Add

Rounding mode

Select the rounding mode for fixed-point operations. If Accumulator Data Type is Inherit: Inherit via internal rule and Output Data Type is Inherit: Same as accumulator, the value of Rounding mode does not affect the numerical results.

Note The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when both of the following conditions exist:

- **Accumulator data type** is `Inherit:  Inherit via internal rule`

- **Output data type** is `Inherit:  Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations.

Vector (V)

Use this parameter to specify the word and fraction lengths for the elements of the vector, *V*. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit:  Same word length as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Note The **Vector (V)** parameter on the **Data Types** pane is only visible when you select `Dialog parameter` for the **Vector (V) source** parameter on the **Main** pane of the block mask. When the vector comes in through the block's input port, the data type and scaling of its elements are inherited from the driving block.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-37 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⬚ ›› ⬚ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

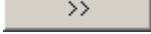See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-37 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟨ >> ⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum
Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum
Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
V	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

See Also

Array-Vector Divide	DSP System Toolbox
Array-Vector Multiply	DSP System Toolbox
Array-Vector Subtract	DSP System Toolbox

Array-Vector Divide

Purpose   Divide array by vector along specified dimension

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3
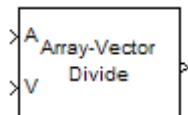
Description   The Array-Vector Divide block divides the values in the specified dimension of the N-dimensional input array *A* by the values in the input vector *V*.

The length of the input *V* must be the same as the length of the specified dimension of *A*. The Array-Vector Divide block divides each element of *V* by the corresponding element along that dimension of *A*.

Consider a 3-dimensional *M*-by-*N*-by-*P* input array *A(i,j,k)* and a *N*-by-1 input vector *V*. When the **Divide along dimension** parameter is set to 2, the output of the block *Y(i,j,k)* is

$$Y(i,j,k) = \frac{A(i,j,k)}{V(j)}$$

where

$$1 \le i \le M$$
$$1 \le j \le N$$
$$1 \le k \le P$$

The output of the Array-Vector Divide block is the same size as the input array, *A*. This block accepts real and complex floating-point and fixed-point input arrays, and real floating-point and fixed-point input vectors.

Fixed-Point Data Types

The following diagram shows the data types used within the Array-Vector Divide block for fixed-point signals.
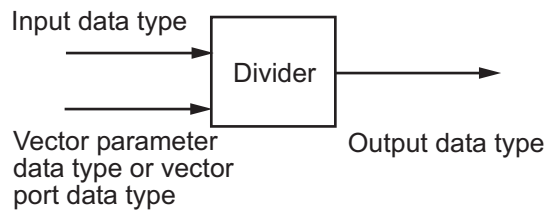
When you specify the vector *V* on the **Main** pane of the block mask, you must specify the data type and scaling properties of its elements in the **Vector (V)** parameter on the **Data Types** tab. When the vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

You can set the vector and output data types in the block dialog as discussed below.

Dialog Box

The **Main** pane of the Array-Vector Divide block dialog appears as follows.

Array-Vector Divide

Function Block Parameters: Array-Vector Divide

Array-Vector Divide

Divide the input array A by the elements of vector V along the specified dimension. In the two-dimensional case, this is equivalent to dividing a full matrix (A) by a diagonal (V). Note that unoriented input signals are treated as oriented column vectors. The output of this block is always oriented.

When you specify the vector V on the block dialog and the fixed-point data type mode is set to "Specify word length," the fixed-point scaling is automatically set for you. In these cases, the scaling is set to the best possible precision given the real-world values and word length of V. This is equivalent to the "Best Precision: Matrix-wise" scaling option used in some fixed-point Simulink blocks.

| Main | Data Types |

Parameters

Divide along dimension: 1

Vector (V) source: Input port

OK   Cancel   Help   Apply

Divide along dimension

Specify the dimension along which to divide the input array *A* by the elements of vector *V*.

Vector (V) source

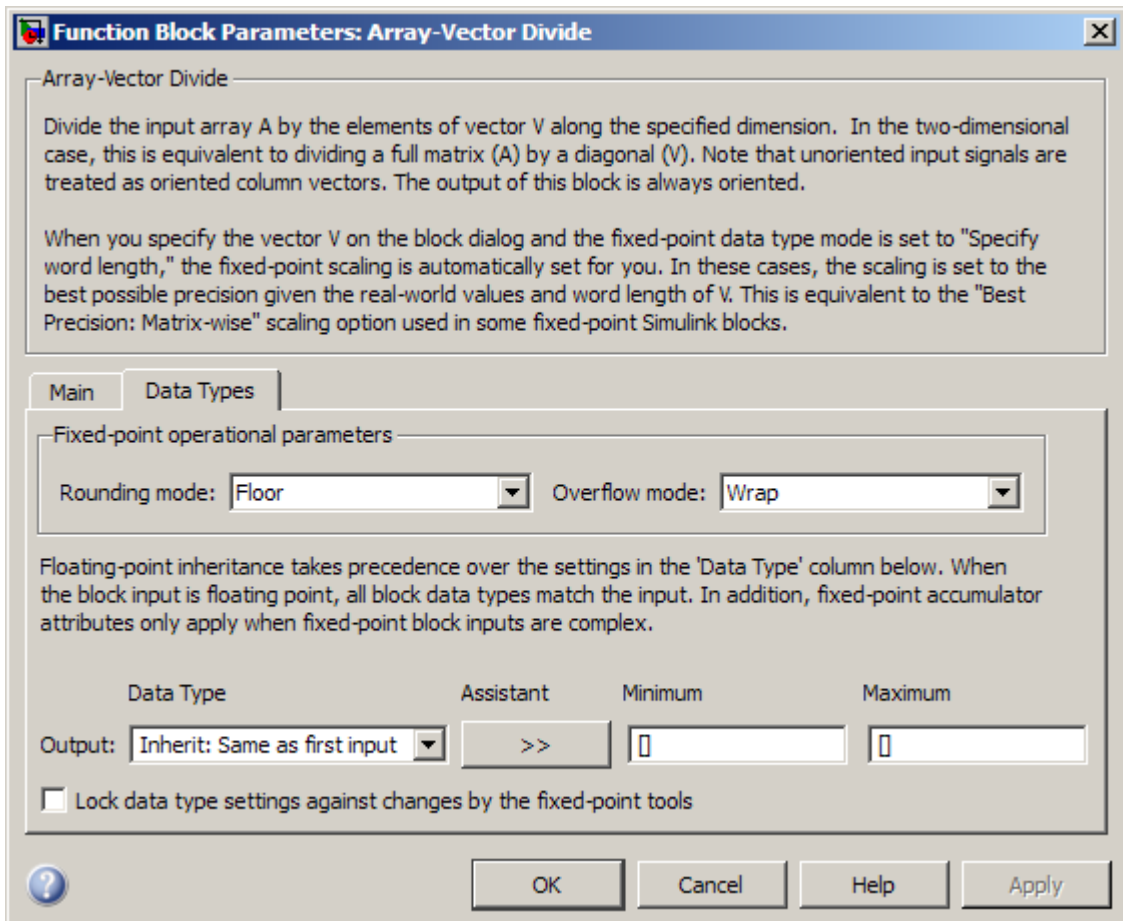Specify the source of the vector, *V*. The vector can come from the Input port or from a Dialog parameter.

Vector (V)

Specify the vector, *V*. This parameter is visible only when you select Dialog parameter for the **Vector (V) source** parameter.

The **Data Types** pane of the Array-Vector Divide block dialog appears as follows.

Array-Vector Divide

Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Vector (V)

Use this parameter to specify the word and fraction lengths for the elements of the vector, *V*. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Note The **Vector (V)** parameter on the **Data Types** pane is only visible when you select `Dialog parameter` for the **Vector (V) source** parameter on the **Main** pane of the block mask. When the vector comes in through the block's input port, the data type and scaling of its elements are inherited from the driving block.

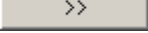**Output data type**

Specify the output data type. See "Fixed-Point Data Types" on page 2-46 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same as first input

- An expression that evaluates to a valid data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Array-Vector Divide

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
V	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

Array-Vector Add	DSP System Toolbox
Array-Vector Multiply	DSP System Toolbox
Array-Vector Subtract	DSP System Toolbox

Purpose   Multiply array by vector along specified dimension

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3
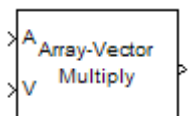
Description   The Array-Vector Multiply block multiplies the values in the specified
dimension of the N-dimensional input array A by the values in the
input vector V.

The length of the input V must be the same as the length of the specified
dimension of A. The Array-Vector Multiply block multiplies each
element of V by the corresponding element along that dimension of A.

Consider a 3-dimensional M-by-N-by-P input array $A(i,j,k)$ and a N-by-1
input vector V. When the **Multiply along dimension** parameter is set
to 2, the output of the block $Y(i,j,k)$ is

$$Y(i, j, k) = A(i, j, k) * V(j)$$

where

$$1 \leq i \leq M$$
$$1 \leq j \leq N$$
$$1 \leq k \leq P$$

The output of the Array-Vector Multiply block is the same size as the
input array, A. This block accepts real and complex floating-point and
fixed-point inputs.

Fixed-Point Data Types

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

Array-Vector Multiply
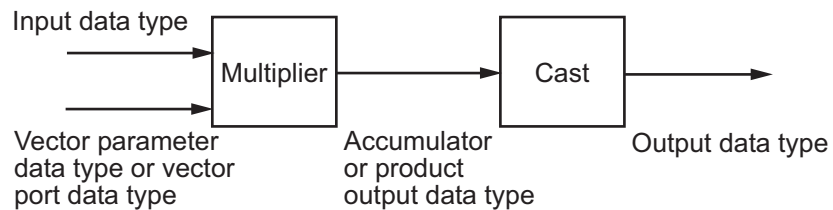
Input data type

Multiplier

Cast

Output data type

Vector parameter
data type or vector
port data type

Accumulator
or product
output data type

When you specify the vector *V* on the **Main** pane of the block mask, you must specify the data type and scaling properties of its elements in the **Vector (V)** parameter on the **Data Types** tab. When the vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types".

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

Dialog Box

The **Main** pane of the Array-Vector Multiply block dialog appears as follows.
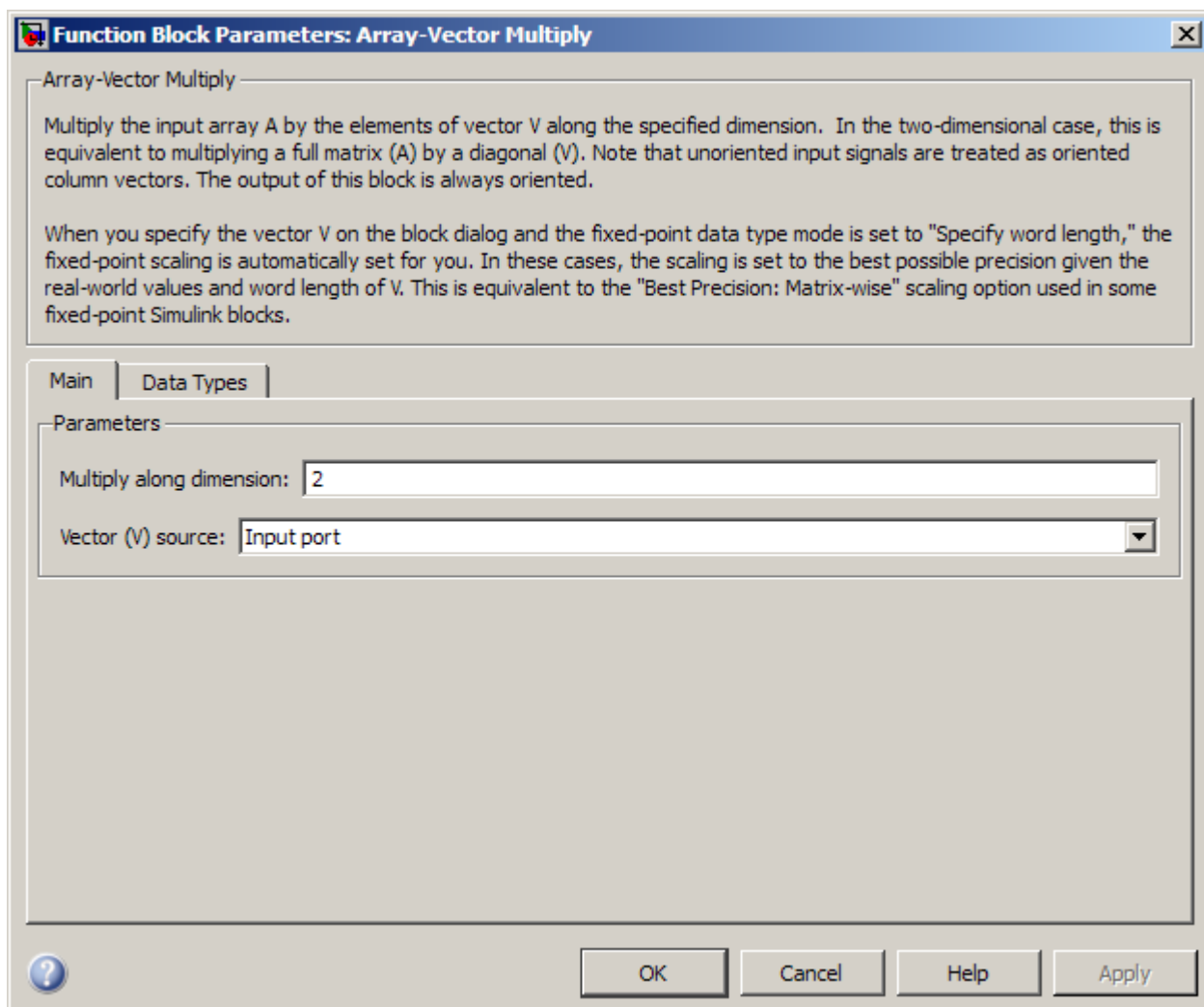
Multiply along dimension

Specify the dimension along which to multiply the input array *A* by the elements of vector *V*.

Array-Vector Multiply

Vector (V) source

> Specify the source of the vector, *V*. The vector can come from the `Input port` or from a `Dialog parameter`.

Vector (V)

> Specify the vector, *V*. This parameter is visible only when you select `Dialog parameter` for the **Vector (V) source** parameter.

The **Data Types** pane of the Array-Vector Multiply block dialog appears as follows.

Function Block Parameters: Array-Vector Multiply ⊠

Array-Vector Multiply

Multiply the input array A by the elements of vector V along the specified dimension. In the two-dimensional case, this is equivalent to multiplying a full matrix (A) by a diagonal (V). Note that unoriented input signals are treated as oriented column vectors. The output of this block is always oriented.

When you specify the vector V on the block dialog and the fixed-point data type mode is set to "Specify word length," the fixed-point scaling is automatically set for you. In these cases, the scaling is set to the best possible precision given the real-world values and word length of V. This is equivalent to the "Best Precision: Matrix-wise" scaling option used in some fixed-point Simulink blocks.

Main	Data Types

Fixed-point operational parameters

Rounding mode: [Floor ▼]   Overflow mode: [Wrap ▼]

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input. In addition, fixed-point accumulator attributes only apply when fixed-point block inputs are complex.

	Data Type	Assistant	Minimum	Maximum
Product output:	Inherit: Inherit via internal rule ▼	>>		
Accumulator:	Inherit: Inherit via internal rule ▼	>>		
Output:	Inherit: Same as product output ▼	>>	[0]	[0]

☐ Lock data type settings against changes by the fixed-point tools

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

Array-Vector Multiply

Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode
  Select the rounding mode for fixed-point operations.

Overflow mode
  Select the overflow mode for fixed-point operations.

Vector (V)
  Use this parameter to specify the word and fraction lengths for the elements of the vector, *V*. You can set this parameter to:

  • A rule that inherits a data type, for example, `Inherit: Same word length as input`

  • An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

  Click the **Show data type assistant** button $\boxed{\quad \gg \quad}$ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

  See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Note The **Vector (V)** parameter on the **Data Types** pane is only visible when you select `Dialog parameter` for the **Vector (V) source** parameter on the **Main** pane of the block mask. When the vector comes in through the block's input port, the data type and scaling of its elements are inherited from the driving block.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-53 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-53 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

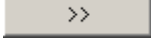See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Array-Vector Multiply

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-53 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
V	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

Array-Vector Add	DSP System Toolbox
Array-Vector Divide	DSP System Toolbox
Array-Vector Subtract	DSP System Toolbox

Array-Vector Subtract

Purpose   Subtract vector from array along specified dimension

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

Description
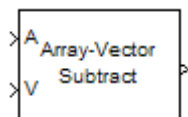
The Array-Vector Subtract block subtracts the values in the input vector *V* from the values in the specified dimension of the N-dimensional input array *A*.

The length of the input *V* must be the same as the length of the specified dimension of *A*. The Array-Vector Subtract block subtracts each element of *V* from the corresponding element along that dimension of *A*.

Consider a 3-dimensional *M*-by-*N*-by-*P* input array *A(i,j,k)* and a *N*-by-1 input vector *V*. When the **Subtract along dimension** parameter is set to 2, the output of the block *Y(i,j,k)* is

$$Y(i,j,k) = A(i,j,k) - V(j)$$

where

$$1 \le i \le M$$
$$1 \le j \le N$$
$$1 \le k \le P$$

The output of the Array-Vector Subtract block is the same size as the input array, *A*. This block accepts real and complex floating-point and fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Array-Vector Subtract block for fixed-point signals.
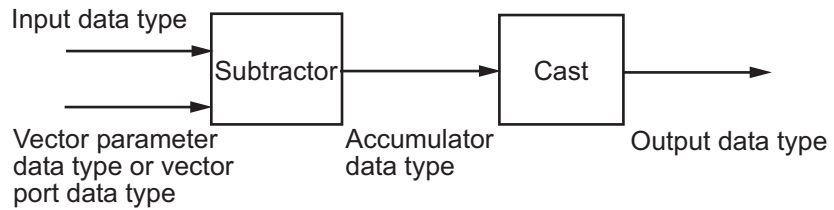
When you specify the vector *V* on the **Main** pane of the block mask, you must specify the data type and scaling properties of its elements in the **Vector (V)** parameter on the **Data Types** tab. When the vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

The output of the subtractor is in the accumulator data type.

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

Dialog Box

The **Main** pane of the Array-Vector Subtract block dialog appears as follows.

Array-Vector Subtract
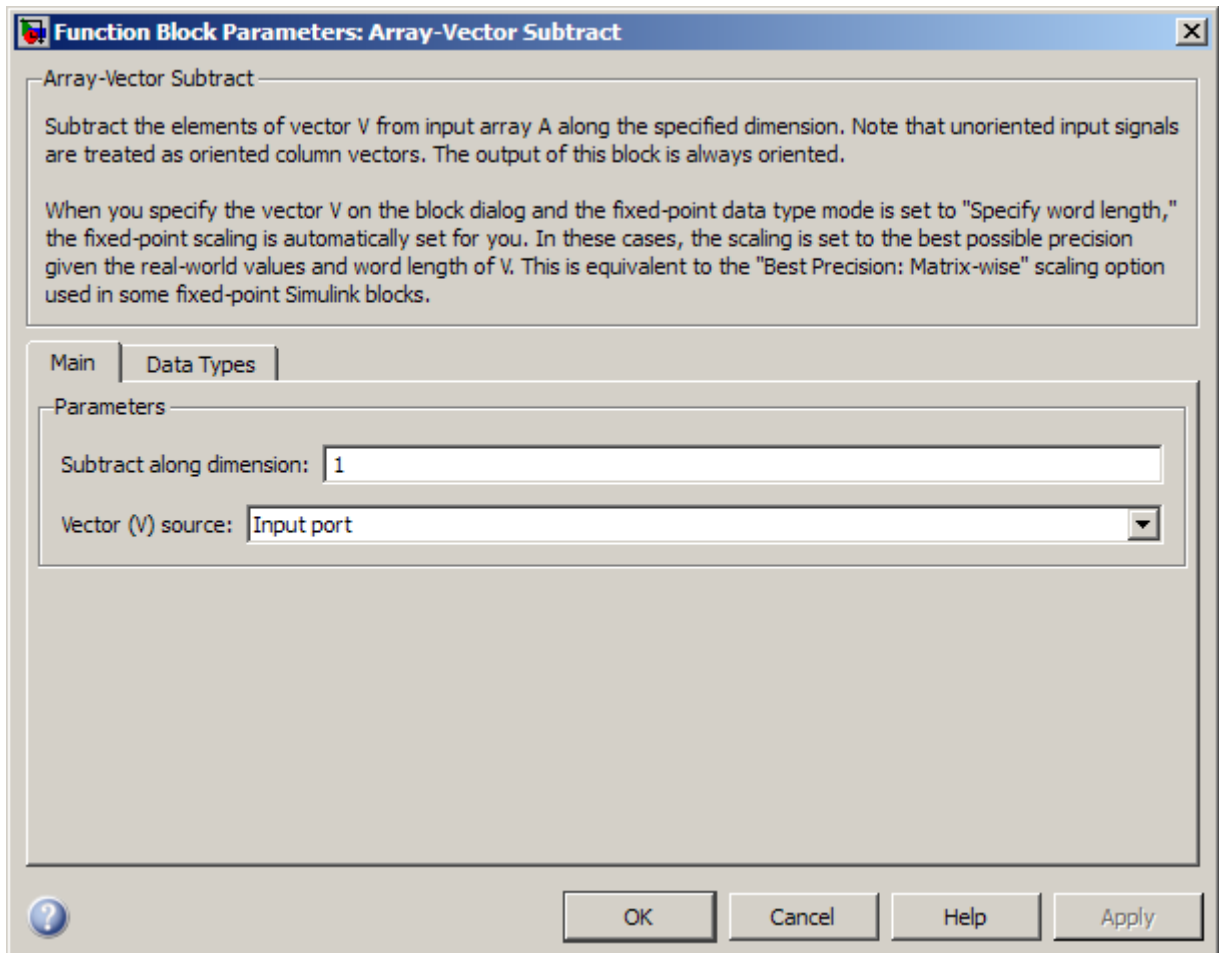
Subtract along dimension

Specify the dimension along which to subtract the elements of vector *V* from the input array *A*.

Vector (V) source

Specify the source of the vector, *V*. The vector can come from the Input port or from a Dialog parameter.

Vector (V)

Specify the vector, *V*. This parameter is visible only when you select Dialog parameter for the **Vector (V) source** parameter.

The **Data Types** pane of the Array-Vector Subtract block dialog appears as follows.

Array-Vector Subtract

Subtract the elements of vector V from input array A along the specified dimension. Note that unoriented input signals are treated as oriented column vectors. The output of this block is always oriented.

When you specify the vector V on the block dialog and the fixed-point data type mode is set to "Specify word length," the fixed-point scaling is automatically set for you. In these cases, the scaling is set to the best possible precision given the real-world values and word length of V. This is equivalent to the "Best Precision: Matrix-wise" scaling option used in some fixed-point Simulink blocks.

Main   **Data Types**

Fixed-point operational parameters

Rounding mode: | Floor   ▼   Overflow mode: | Wrap   ▼

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input. In addition, fixed-point accumulator attributes only apply when fixed-point block inputs are complex.

	Data Type	Assistant	Minimum	Maximum
Accumulator:	Inherit: Inherit via internal rule ▼	>>		
Output:	Inherit: Same as accumulator ▼	>>	0	0

☐ Lock data type settings against changes by the fixed-point tools

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

Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

> **Note** The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when both of the following conditions exist:
>
> - **Accumulator data type** is Inherit: Inherit via internal rule
>
> - **Output data type** is Inherit: Same as accumulator
>
> With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations.

Vector (V)

Use this parameter to specify the word and fraction lengths for the elements of the vector, *V*. You can set this parameter to:

- A rule that inherits a data type, for example, Inherit: Same word length as input

- An expression that evaluates to a valid data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Array-Vector Subtract

> **Note** The **Vector (V)** parameter on the **Data Types** pane is only visible when you select `Dialog parameter` for the **Vector (V) source** parameter on the **Main** pane of the block mask. When the vector comes in through the block's input port, the data type and scaling of its elements are inherited from the driving block.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-62 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button <kbd>>></kbd> to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-62 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit:   Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum
Specify the minimum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum
Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
V	• Double-precision floating point
	• Single-precision floating point

Array-Vector Subtract

Port	Supported Data Types
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

See Also

Array-Vector Add	DSP System Toolbox
Array-Vector Divide	DSP System Toolbox
Array-Vector Multiply	DSP System Toolbox

Purpose   Design audio weighting filter

Library   Filtering / Filter Designs

dspfdesign

Description   This block brings the filter design capabilities of the filterbuilder function to the Simulink environment.

> Audio Weighting >

Audio Weighting Filter
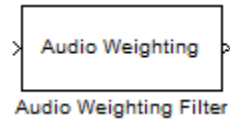
Dialog Box   See "Audio Weighting Filter Design Dialog Box — Main Pane" on page 7-672 for more information about the parameters of this block. The **Data Types** and **Code** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Audio Weighting Filter
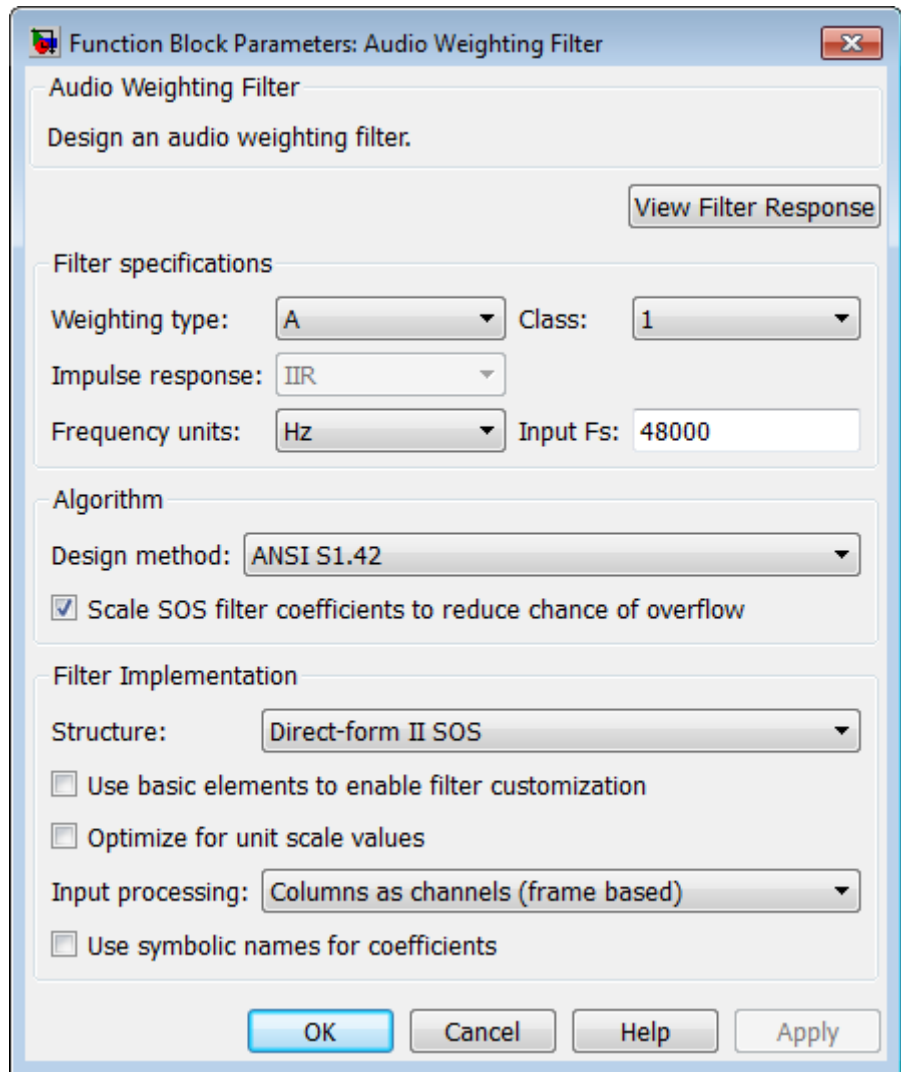
Function Block Parameters: Audio Weighting Filter

Audio Weighting Filter

Design an audio weighting filter.

View Filter Response

Filter specifications

Weighting type: A   Class: 1

Impulse response: IIR

Frequency units: Hz   Input Fs: 48000

Algorithm

Design method: ANSI S1.42

☑ Scale SOS filter coefficients to reduce chance of overflow

Filter Implementation

Structure: Direct-form II SOS

☐ Use basic elements to enable filter customization

☐ Optimize for unit scale values

Input processing: Columns as channels (frame based)

☐ Use symbolic names for coefficients

OK   Cancel   Help   Apply

View Filter Response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Weighting type

The weighting type defines the frequency response of the filter. The valid weighting types for this filter are `A`, `C` , `C-message`, `ITU-R 468 4`, and `ITU-T 0.41`. For definitions of the available weighting types, see the `fdesign.audioweighting` reference page.

Class

The filter class describes the frequency-dependent tolerances specified in the relevant standards [1], [2]. There are two possible class values: `1` and `2`. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in `fvtool` for the analysis of the filter design. The default value of this parameter is `1`.

The filter class is only applicable for `A` weighting and `C` weighting filters.

Audio Weighting Filter

Impulse response

Specify the impulse response type as one of `IIR` or `FIR`. For A, C , C-message, and ITU-R 468–4 filter, `IIR` is the only option. For a ITU-T 0.41 weighting filter, `FIR` is the only option.

Frequency units

Specify the frequency units as Hertz (Hz), kilohertz (kHz), megahertz (MHz), or gigahertz (GHz). Normalized frequency designs are not supported for audio weighting filters. The default value of this parameter is `Hz`.

Input Fs

Specify the input sampling frequency. The units correspond to the setting of the **Frequency units** parameter.

Algorithm

Design Method

Valid design methods depend on the weighting type. For type A and C weighting filters, the only valid design type is `ANSI S1.42`. This is an IIR design method that follows ANSI standard S1.42–2001. For a C message filter, the only valid design method is `Bell 41009`, which is an IIR design method following the Bell System Technical Reference PUB 41009. For a ITU-R 468–4 weighting filter, you can design an IIR or FIR filter. If you choose an IIR design, the design method is `IIR least p-norm`. If you choose an FIR design, the design method choices are `Equirriple` or `Frequency Sampling`. For an ITU-T 0.41 weighting filter, the available FIR design methods are `Equirriple` or `Frequency Sampling`.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Filter Implementation

Structure

> For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. For audio weighting IIR filter designs, you can choose direct form I or II biquad (SOS). You can also choose to implement these structures in transposed form.
>
> For FIR designs, you can choose a direct form, direct-form transposed, direct-form symmetric, or direct-form asymmetric structure.

Use basic elements to enable filter customization

> Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.
>
> The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:
>
> - **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
>
> - **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
>
> - **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.
>
> - **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

> Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

For more information about sample- and frame-based processing, see "Sample- and Frame-Based Concepts".

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

References

[1] *American National Standard Design Response of Weighting Networks for Acoustical Measurements*, ANSI S1.42-2001, Acoustical Society of America, New York, NY, 2001.

[2] *Electroacoustics Sound Level Meters Part 1: Specifications*, IEC 61672-1, First Edition 2002-05.

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Supported Data Types

See Also   `fdesign.audioweighting` | `filterbuilder` | `fvtool`

How To   • Audio Weighting Filters Example

Autocorrelation

Purpose   Autocorrelation of vector or matrix input

Library   Statistics

dspstat3

Description   The Autocorrelation block computes the autocorrelation along the first dimension of an N-D input array. The block accepts fixed-point signals when you set the **Computation domain** to Time.

When the input to the Autocorrelation block is an *M*-by-*N* matrix *u*, the output, *y*, is an (*l*+1)-by-*N* matrix whose *j*th column has elements

$$y_{i,j} = \sum_{k=0}^{M-l-1} u_{k,j}^* u_{(k+i),j} \qquad 0 \le i \le l$$

where * denotes the complex conjugate, and *l* represents the maximum lag. $y_{0,j}$ is the zero-lag element in the *j*th column. When you select **Compute all non-negative lags**, *l*=*M*–1. Otherwise, *l* is the nonnegative integer value you specify for the **Maximum non-negative lag (less than input length)** parameter.

Similarly, for an N-D input array, the block outputs an N-D array, where the size of the first dimension is *l*+1, and the sizes of all other dimensions match those of the input array. For example, when the input is an *M*-by-*N*-by-*P* array, the Autocorrelation block outputs an (*l*+1)-by-*N*-by-*P* array.

Fixed-Point Data Types

The following diagrams show the data types used within the Autocorrelation block for fixed-point signals (time domain only).

Signal flow when Scaling is "None"

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

Input
data type

Multiplier → Cast → Adder → Cast → Output
data type

$\frac{1}{z}$

Accumulator
or Product
output data type

Accumulator
data type
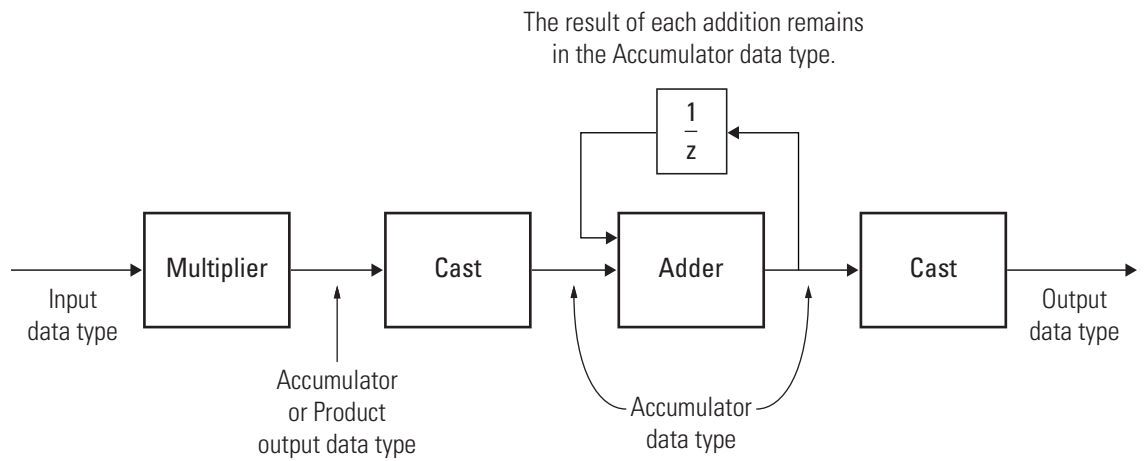
Signal flow when Scaling is other than "None"

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

Input
data type

Multiplier → Cast → Adder → Divider → Output
data type

$\frac{1}{z}$

Accumulator
or Product
output data type

Accumulator
data type

2-79
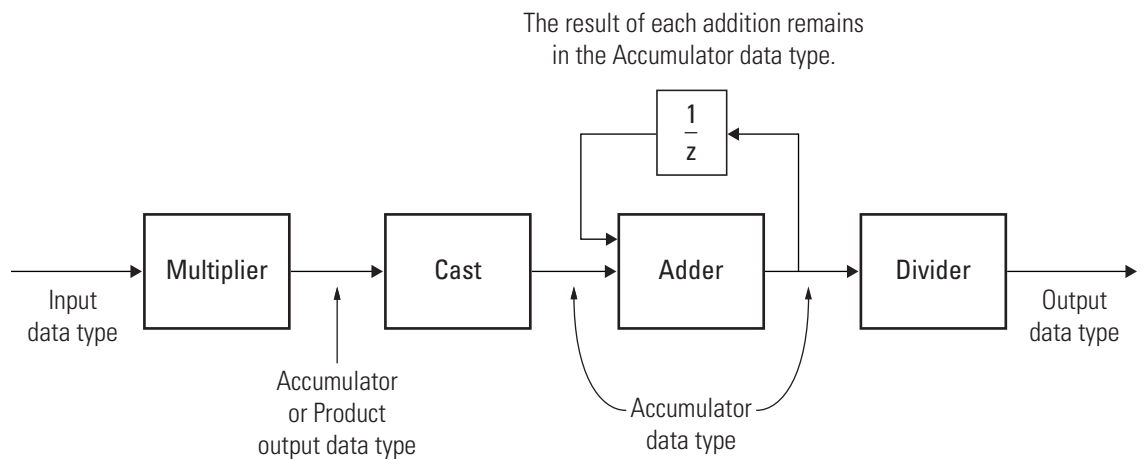
Autocorrelation

You can set the product output, accumulator, and output data types on the **Data Types** pane of the block dialog as discussed in the next section.

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

Dialog Box

The **Main** pane of the Autocorrelation block dialog appears as follows.
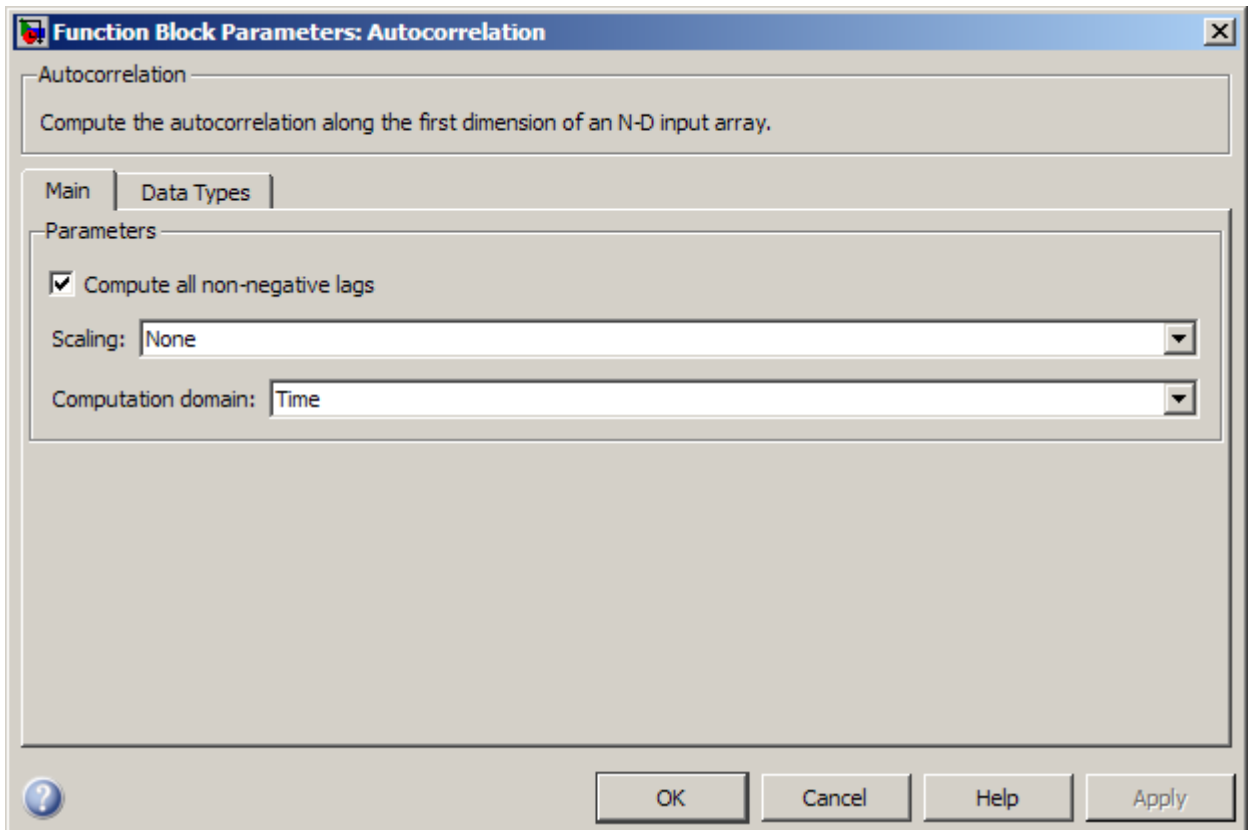
Compute all non-negative lags
When you select this check box, the block computes the
autocorrelation over all nonnegative lags in the range [0,
length(input)-1]. When you clear the check box, the block
computes the autocorrelation using lags in the range [0, *l*], where
l is the value you specify for the **Maximum non-negative lag
(less than input length)** parameter.

Autocorrelation

Maximum non-negative lag (less than input length)

Specify the maximum positive lag, *l*, for the autocorrelation. This parameter is available only when you clear the **Compute all non-negative lags** check box.

Scaling

This parameter controls the scaling that the block applies 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.

$$y_{0,j} = 1$$

Computation domain

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

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

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

> **Note** This parameter must be set to `Time` for fixed-point signals.

The **Data Types** pane of the Autocorrelation block dialog appears as follows.

Autocorrelation

> **Note** Fixed-point signals are only supported for the time domain. To use the parameters on this pane, make sure Time is selected for the **Computation domain** parameter on the **Main** pane.

Rounding mode

Select the rounding mode for fixed-point operations.

> **Note** The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when all the following conditions exist:
>
> • **Product output data type** is Inherit: Inherit via internal rule
>
> • **Accumulator data type** is Inherit: Inherit via internal rule
>
> • **Output data type** is Inherit: Same as accumulator
>
> With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-78 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

• A rule that inherits a data type, for example, Inherit: Inherit via internal rule

• An expression that evaluates to a valid data type, for example, fixdt([],16,0)

Click the **Show data type assistant** button $\boxed{\quad \gg \quad}$ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-78 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button $\boxed{\quad \gg \quad}$ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.
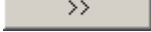
Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-78 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ⬚ `>>` to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum
Specify the minimum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum
Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

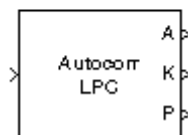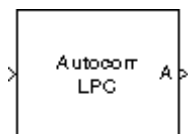|	
Correlation	DSP System Toolbox
xcorr	Signal Processing Toolbox

Autocorrelation LPC

Purpose   Determine coefficients of Nth-order forward linear predictors

Library   Estimation / Linear Prediction

dsplp

Description

The Autocorrelation LPC block determines the coefficients of an *N-step forward linear predictor* for the time-series in each length-*M* input channel, *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 for each channel as polynomial coefficients, reflection coefficients, or both. It can also output the prediction error power for each channel. The input *u* can be a scalar, unoriented vector, column vector, sample-based row vector, or a matrix. Frame-based row vectors are not valid inputs. The block treats all *M*-by-*N* matrix inputs as *N* channels of length *M*.

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

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

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

When **Output(s)** is set to K, port K is enabled. For each channel, 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 set of prediction coefficients for each channel.

When you select **Output prediction error power (P)**, port P is enabled. The prediction error power is output at port P as a vector whose length is the number of input channels.

Algorithm The Autocorrelation LPC block computes the least squares solution to

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

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

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

Solving the least squares problem via the normal equations

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

leads to the system of equations

Autocorrelation LPC

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

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

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

Function Block Parameters: Autocorrelation LPC

Autocorrelation LPC (mask) (link)

Output the coefficients of an Nth order forward linear predictor such that the sum of the squares of the errors is minimized (using the autocorrelation LPC method). The prediction coefficients are given in A (polynomial coefficients) and/or K (reflection coefficients). The prediction error power is given in the optional P output.

Parameters

Output(s): A

☐ Output prediction error power (P)

☐ Inherit prediction order from input dimensions

Prediction order (N):

1

OK   Cancel   Help   Apply

Dialog Box

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

Output prediction error power (P)
> When selected, enables port P, which outputs the output prediction error power.

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

Prediction order (N)
Specify the prediction order, *N*, which must be a scalar. This parameter is disabled when you select the **Inherit prediction order from input dimensions** parameter.

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

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

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

Supported Data Types
- Double-precision floating point
- Single-precision floating point

See Also

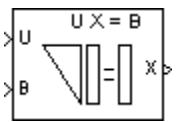|	
Autocorrelation	DSP System Toolbox
Levinson-Durbin	DSP System Toolbox
Yule-Walker Method	DSP System Toolbox
lpc	Signal Processing Toolbox

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

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers

dspsolvers

Description   The Backward Substitution block solves the linear system *UX=B* by simple backward substitution of variables, where:

- *U* is the upper triangular *M*-by-*M* matrix input to the U port.

- *B* is the *M*-by-*N* matrix input to the B port.

The *M*-by-*N* output matrix *X* is the solution of the equations. The block does not check the rank of the inputs.

The block uses only the elements in the *upper triangle* of input *U* and ignores the lower elements. When you select the **Input U is unit-upper triangular** check box, the block assumes the elements on the diagonal of *U* are 1s. 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.

The block treats a length-*M* vector input at port B as an *M*-by-1 matrix.

Fixed-Point Data Types

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

Backward Substitution

When input U is not unit-upper triangular:
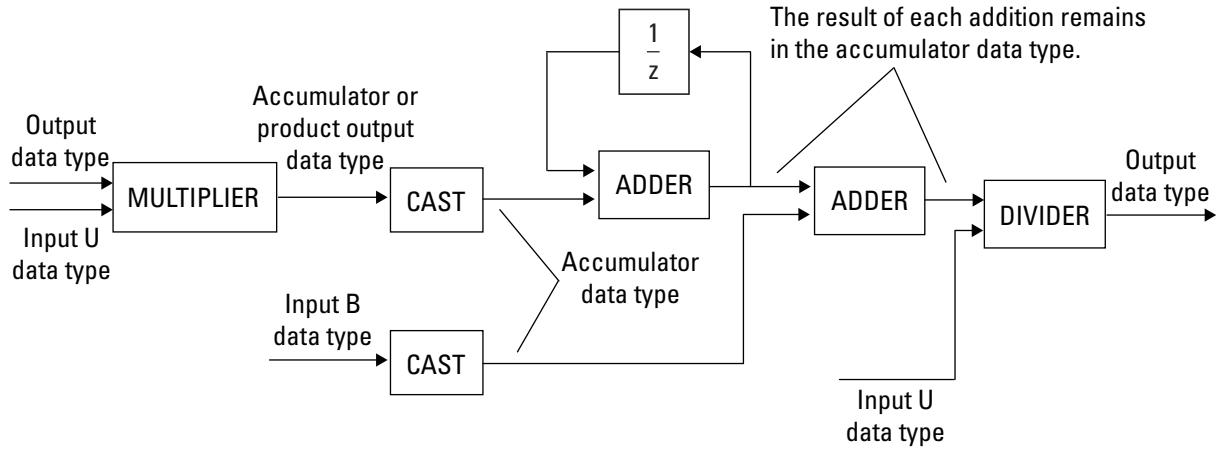
When input U is unit-upper triangular:
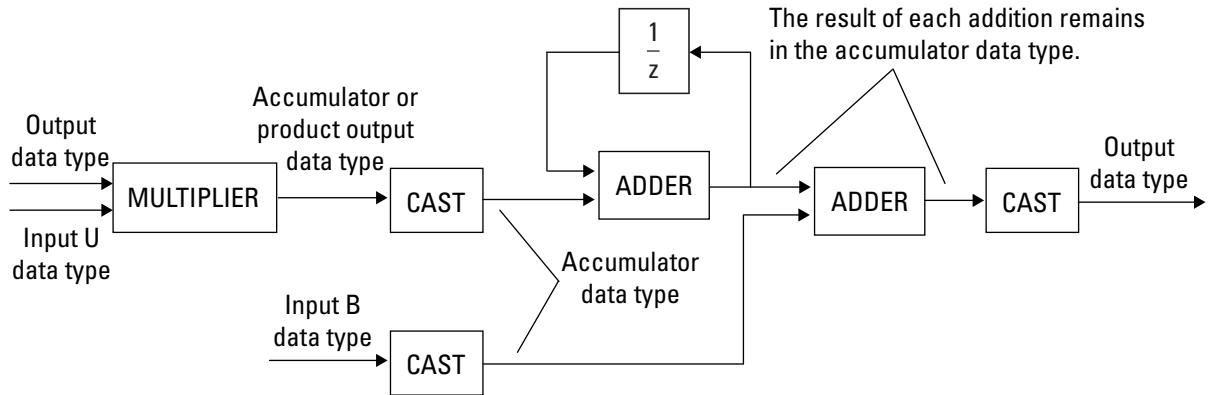
You can set the product output, accumulator, and output data types in the block dialog as discussed in the following section.

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

Dialog Box

The **Main** pane of the Backward Substitution block dialog box appears as follows.

Backward Substitution
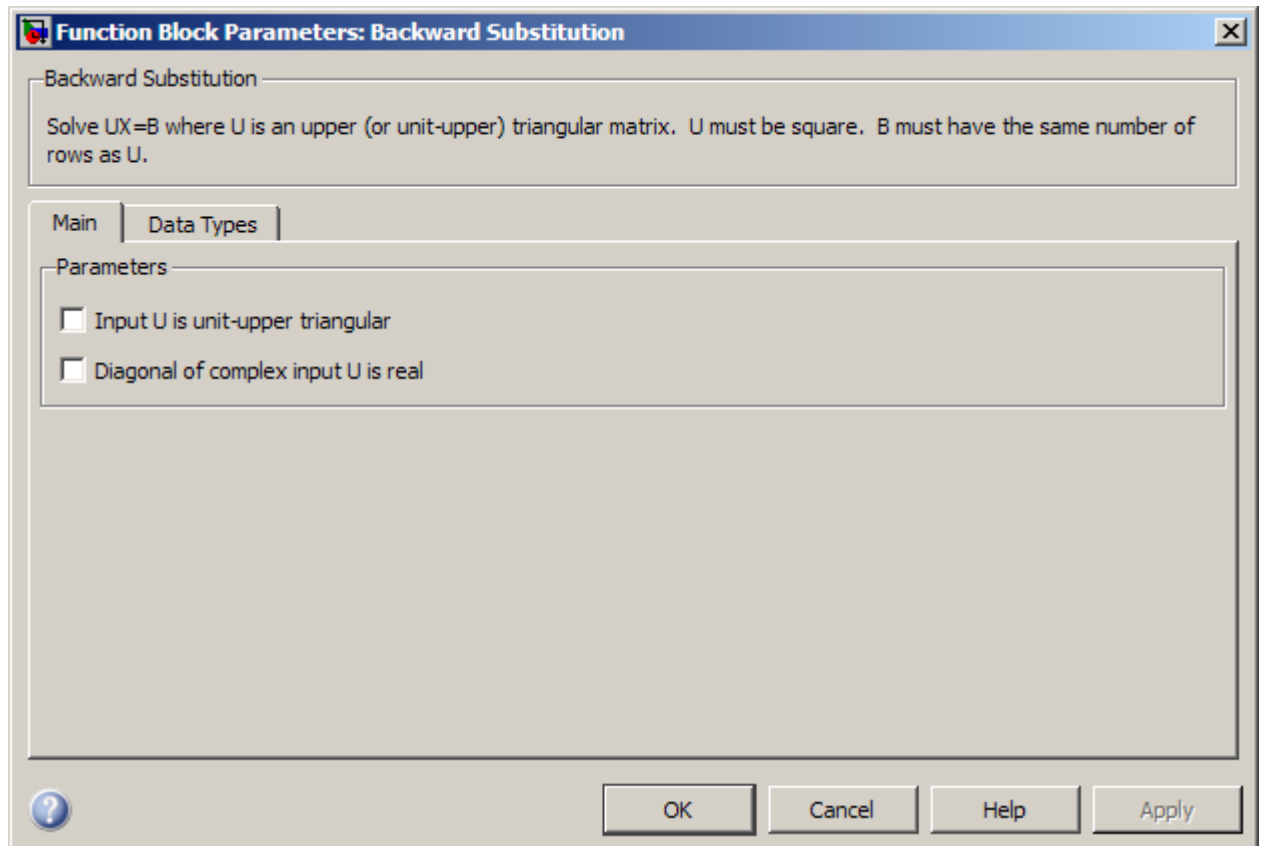
Input U is unit-upper triangular

Select this check box only when all elements on the diagonal of *U* have a value of 1. When you do so, the block optimizes its behavior by skipping an unnecessary divide operation.

Do not select this check box if there are any elements on the diagonal of *U* that do not have a value of 1. When you clear the **Input U is unit-upper triangular** check box, the block always performs the necessary divide operation.

Diagonal of complex input U is real

Select to optimize simulation speed when the diagonal elements of complex input U are real. This parameter is only visible when **Input U is unit-upper triangular** is not selected.

Note When U is a complex fixed-point signal, you must select either **Input U is unit-upper triangular** or **Diagonal of complex input U is real**. In such a case, any imaginary part of the diagonal of U is ignored.

The **Data Types** pane of the Backward Substitution block dialog appears as follows.

Backward Substitution

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

Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.
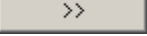
Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-93 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-93 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-93 for illustrations depicting the use of the output data type in this block. You can set it to:

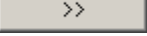- A rule that inherits a data type, for example, `Inherit: Same as first input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
U	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
B	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
X	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

Cholesky Solver	DSP System Toolbox
Forward Substitution	DSP System Toolbox
LDL Solver	DSP System Toolbox
Levinson-Durbin	DSP System Toolbox
LU Solver	DSP System Toolbox
QR Solver	DSP System Toolbox

See "Linear System Solvers" for related information.

Bandpass Filter

Purpose   Design bandpass filter

Library   Filtering / Filter Designs

dspfdesign

Description   This block brings the filter design capabilities of the `filterbuilder`
function to the Simulink environment.

```
   ┌─────────────┐
 ─>│  Bandpass   │>─
   └─────────────┘
```

Dialog   See "Bandpass Filter Design Dialog Box — Main Pane" on page 7-675
Box   for more information about the parameters of this block. The **Data
                   Types** and **Code Generation** panes are not available for blocks in the
                   DSP System Toolbox Filter Designs library.

Bandpass Filter

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response
Select either FIR or IIR from the drop-down list. FIR is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode
Select Minimum (the default) or Specify. Selecting Specify enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to IIR, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

Order
Enter the filter order. This option is enabled only if you set the **Order mode** to Specify.

Denominator order

Select this check box to specify a different denominator order. This option is enabled only if you set the **Impulse response** to IIR and the **Order mode** to Specify.

Filter type

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

Bandpass Filter

In the figure, regions between specification values such as Fstop1 and Fpass1 represent transition regions where the filter response is not constrained.

Frequency constraints

When **Order mode** is Specify, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- Passband and stopband edges — Define the filter by specifying the frequencies for the edges for the stop- and passbands.

- Passband edges — For IIR filters, define the filter by specifying frequencies for the edges of the passband.

- Stopband edges — For IIR filters, define the filter by specifying frequencies for the edges of the stopbands.

- 3 dB points — For IIR filters, define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point three decibels below the passband value.

- 3 dB points and passband width — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.

- `3 dB points and stopband widths` — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

- `6 dB points` — For FIR filters, define the filter response by specifying the locations of the 6 dB points. The 6 dB point is the frequency for the point six decibels below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fstop1

Enter the frequency at the edge of the end of the first stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass1

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you selected for **Frequency units**.

Fpass2

Enter the frequency at the edge of the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop2

Enter the frequency at the edge of the start of the second stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

F3dB1

When **Frequency constraints** is 3 dB points, 3 dB points and passband width, or 3 dB points and stopband width, specify the lower-frequency 3 dB point.

F3dB2

When **Frequency constraints** is 3 dB points, 3 dB points and passband width, or 3 dB points and stopband width, specify the higher-frequency 3 dB point.

F6dB1

When **Frequency constraints** is 6 dB points, specify the lower-frequency 6 dB point.

F6dB2

When **Frequency constraints** is 6 dB points, specify the higher-frequency 6 dB point.

Passband width

When **Frequency constraints** is 3 dB points and passband width, specify the width of the passband, in units corresponding to the **Frequency units** parameter.

Stopband width

When **Frequency constraints** is 3 dB points and stopband width, specify the width of the stopband, in units corresponding to the **Frequency units** parameter.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude constraints

This option is only available when you specify the order of your filter design. The options for **Magnitude constraints** depend

on the value of the **Frequency constraints**. When you set the **Frequency constraints** parameter to Unconstrained, **Magnitude constraints** must also be set to Unconstrained. When **Frequency constraints** is not set to Unconstrained, some combination of the following options will be available for the **Magnitude constraints** parameter: Unconstrained, Passband ripple, Passband ripple and stopband attenuation or Stopband attenuation.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications.

From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in dB (decibels). This is the default setting.

- Squared — Specify the magnitude in squared units.

Astop1

Enter the filter attenuation in the first stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Bandpass Filter

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is `Equiripple`.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Note Generally, **Minimum order** designs are not available for IIR filters.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Bandpass Filter

Use basic elements to enable filter customization
Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values
Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing
Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

> **Note** The Inherited (this choice will be removed   see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- Enforce single-rate processing — When you select this option, the block maintains the sample rate of the input.

- Allow multirate processing — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to Elements as channels (sample based).

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

Bandstop Filter

Purpose   Design bandstop filter

Library   Filtering / Filter Designs

dspfdesign

Description   This block brings the filter design capabilities of the filterbuilder
function to the Simulink environment.

Bandstop

Dialog Box   See "Bandstop Filter Design Dialog Box — Main Pane" on page 7-683
for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the
DSP System Toolbox Filter Designs library.

Bandstop Filter

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response
Select either FIR or IIR from the drop-down list. FIR is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode
Select Minimum (the default) or Specify. Selecting Specify enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to IIR, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

Order
Enter the filter order. This option is enabled only if you set the **Order mode** to Specify.

Denominator order

Select this check box to specify a different denominator order. This option is enabled only if you set the **Impulse response** to IIR and the **Order mode** to Specify.

Filter type

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

Bandstop Filter

Frequency constraints

> When **Order mode** is `Specify`, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.
>
> - `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
>
> - `Passband edges` — For IIR filters, define the filter by specifying frequencies for the edges of the passband.
>
> - `Stopband edges` — For IIR filters, define the filter by specifying frequencies for the edges of the stopbands.
>
> - `3 dB points` — For IIR filters, define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point three decibels below the passband value.
>
> - `3 dB points and passband width` — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.
>
> - `3 dB points and stopband widths` — For IIR filters, define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

- `6 dB points` — For FIR filters, define the filter response by specifying the locations of the 6 dB points. The 6 dB point is the frequency for the point six decibels below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

When you design an interpolator, Fs represents the sampling frequency at the filter output rather than the filter input. This option is available only when you set **Filter type** is `interpolator`.

Fpass1

Enter the frequency at the edge of the end of the first passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop1

Enter the frequency at the edge of the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop2

Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass2

Enter the frequency at the edge of the start of the second passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

F3dB1

When **Frequency constraints** is 3 dB points, 3 dB points and passband width, or 3 dB points and stopband width, specify the lower-frequency 3 dB point.

F3dB2

When **Frequency constraints** is 3 dB points, 3 dB points and passband width, or 3 dB points and stopband width, specify the higher-frequency 3 dB point.

F6dB1

When **Frequency constraints** is 6 dB points, specify the lower-frequency 6 dB point.

F6dB2

When **Frequency constraints** is 6 dB points, specify the higher-frequency 6 dB point.

Passband width

When **Frequency constraints** is 3 dB points and passband width, specify the width of the passband, in units corresponding to the **Frequency units** parameter.

Stopband width

When **Frequency constraints** is 3 dB points and stopband width, specify the width of the stopband, in units corresponding to the **Frequency units** parameter.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude constraints

This option is only available when you specify the order of your filter design. The options for **Magnitude constraints** depend on

the value of the **Frequency constraints**. Depending on the value of the **Frequency constraints** parameter, some combination of the following options will be available for the **Magnitude constraints** parameter: Unconstrained, Passband ripple and stopband attenuation, or Constrained bands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Apass1

Enter the filter ripple allowed in the first passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels

Apass2

Enter the filter ripple allowed in the second passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is Equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list

to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed  see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.

- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients
> Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Biquad Filter

Purpose	Model biquadratic IIR (SOS) filters
Library	Filtering / Filter Implementations
	dsparch4

Description

The Biquad Filter block independently filters each channel of the input signal with the specified biquadratic IIR filter. When you specify the filter coefficients in the dialog box, the block implements static filters with fixed coefficients. When you provide the filter coefficients through an input port, you can tune the coefficients during simulation.

The block filters an *M*-by-*N* input matrix as follows:

- When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each column as a separate channel. In this mode, the block creates *M* instances of the same filter, each with its own independent state buffer. Each of the *M* filters process *N* input samples at every Simulink time step.

- When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element as a separate channel. In this mode, the block creates *M*N* instances of the same filter, each with its own independent state buffer. Each filter processes one input sample at every Simulink time step.

The output dimensions always equal those of the input signal. The outputs of this block numerically match the outputs of the `dfilt` object.

The Biquad Filter block supports the Simulink state logging feature. See "States" in the *Simulink User's Guide* for more information.

Coefficient Source and Filter Structures

The Biquad Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box.

- If you select **Dialog parameters**, you enter information about the filter such as structure and coefficients in the block mask. In this

mode, you can choose the following filter structures in the **Filter structure** parameter:

- Direct form I
- Direct form I transposed
- Direct form II
- Direct form II transposed

- If you select **Input port(s)**, you enter information about the filter structure in the block mask using the **Filter structure** parameter, but the filter coefficients come into the block via input ports. The following additional ports appear on the block icon:

  - Num — numerator coefficients
  - Den — denominator coefficients
  - g — scale values

- If you select **Discrete-time filter object (DFILT)**, you specify the filter using a dfilt object. This block supports the following dfilt structures:

  - dfilt.df1sos
  - dfilt.df1tsos
  - dfilt.df2sos
  - dfilt.df2tsos

Specifying the SOS Matrix and Scale Values

The **SOS matrix (Mx6)** is an *M*-by-6 matrix, where *M* is the number of sections in the second-order section filter. Each row of the SOS matrix contains the numerator and denominator coefficients (b_{ik} and a_{ik}) of the corresponding section in the filter.

$$\begin{bmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0M} & b_{1M} & b_{2M} & a_{0M} & a_{1M} & a_{2M} \end{bmatrix}$$

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

The **Scale values** parameter specifies the scale values the block uses between each SOS section. You can specify a real-valued scalar or a vector of length $M+1$.

- If you enter a scalar, the value specifies the gain value before the first section of the second-order filter. The rest of the gain values default to `1`.

- If you enter a vector of $M+1$ values, each value specifies a separate section of the filter. For example, the first element is the first gain value, the second element is the second gain value, and so on.
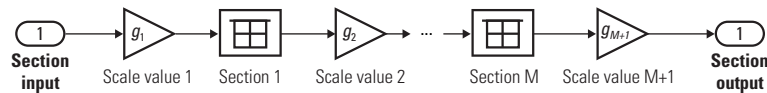
Select the **Optimize unity scale values** check box to optimize your simulation when one or more scale values equal 1. Selecting this option removes the unity gains so that the values are treated like Simulink lines or wires. In some fixed-point cases when there are unity scale values, selecting this parameter also omits certain casts. Refer to "Filter Structure Diagrams" on page 2-148 for more information.

Specifying Initial Conditions

The Biquad Filter block initializes the internal filter states to zero by default. You can optionally use the **Initial conditions** or **Initial conditions on zeros side** and **Initial conditions on poles side** parameters to specify nonzero initial states for the filter delays.

To determine the number of initial conditions you must specify and how to specify them, see the following table on valid initial conditions.

Valid Initial Conditions

Initial Condition	Description
Scalar	The block initializes all delay elements in the filter to the scalar value.
Vector or matrix (for applying different delay elements to each channel)	Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel. Where M is the number of sections and N is the number of input channels: • The vector length must equal the number of delay elements in the filter, $M*2$. • The matrix must have the same number of rows as the number of delay elements in the filter, $(M*2)*N$. The matrix must also have one column for each channel of the input signal.

Fixed-Point Data Types

See the "Filter Structure Diagrams" on page 2-148 section for diagrams showing the data types the biquad filter block uses when processing fixed-point signals.

Examples   Open an example model by typing `ex_biquad_filter_ref` at the MATLAB command line.

Dialog Box   ### Coefficient Source

The Biquad Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box.

• If you select **Dialog parameters**, you enter information about the filter such as structure and coefficients in the block mask. In this

mode, you can choose the following filter structures in the **Filter structure** parameter:

- Direct form I

- Direct form I transposed

- Direct form II

- Direct form II transposed

- If you select **Input port(s)**, you enter information about the filter structure in the block mask using the **Filter structure** parameter, but the filter coefficients come into the block via input ports. The following additional ports appear on the block icon:

  - Num — numerator coefficients

  - Den — denominator coefficients

  - g — scale values

- If you select **Discrete-time filter object (DFILT)**, you specify the filter using a dfilt object. This block supports the following dfilt structures:

  - dfilt.df1sos

  - dfilt.df1tsos

  - dfilt.df2sos

  - dfilt.df2tsos

Specify Filter Characteristics in Dialog

The **Main** pane of the Biquad Filter block dialog appears as follows when **Dialog parameters** is selected in the **Coefficient source** group box.

Function Block Parameters: Biquad Filter

Biquad Filter

Implement a general IIR filter using biquad structures. Biquad implementations of general IIR filters are often preferred due to their desirable numeric properties.

Coefficient source

- ⦿ Dialog parameters
- ○ Input port(s)
- ○ Discrete-time filter object (DFILT)

| Main | Data Types |

Parameters

Filter structure: Direct form II transposed

SOS Matrix (Mx6): [1 0.3 0.4 1 0.1 0.2]

Scale values: 1

Initial conditions: 0

Action when the a0 values of the SOS matrix are not one: Warning

☑ Optimize unity scale values

Input processing: Columns as channels (frame based)

View Filter Response

OK   Cancel   Help   Apply
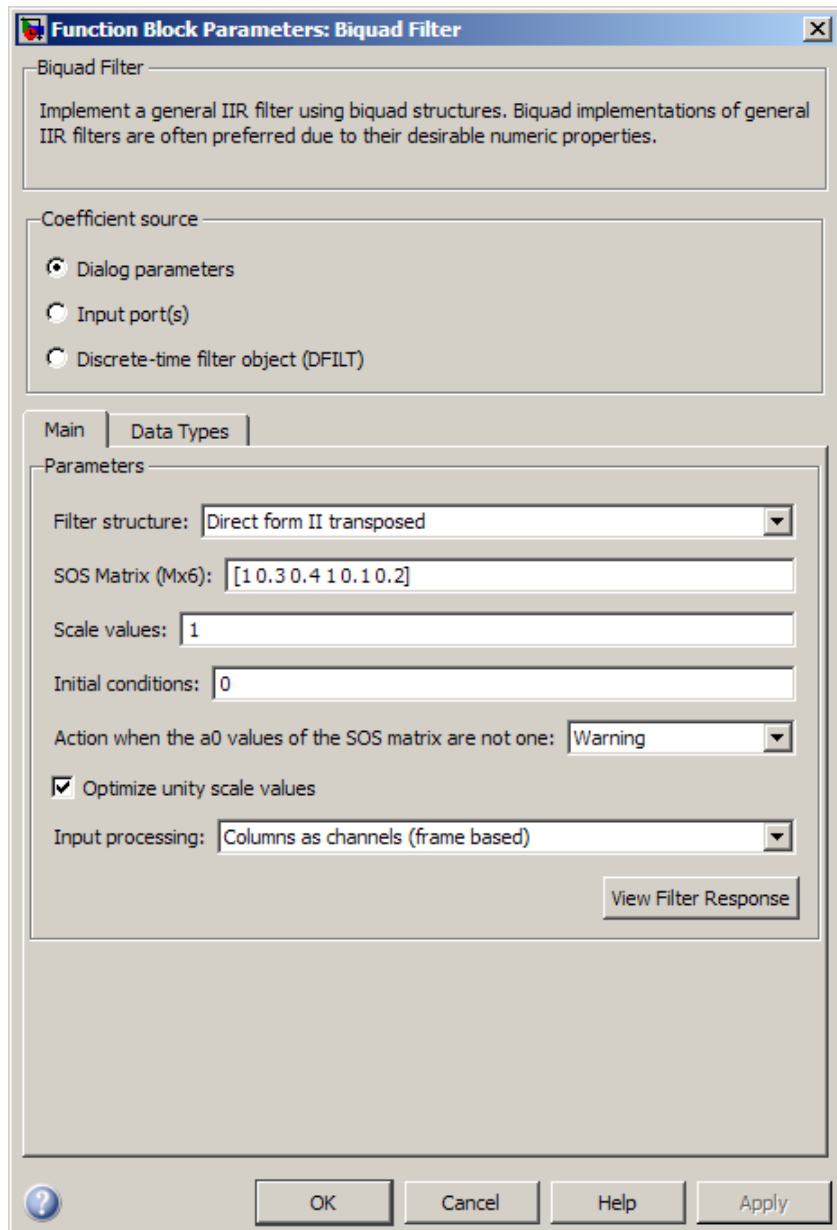
Biquad Filter

Filter structure

Select the filter structure.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected.

SOS Matrix

Specify an M-by-6 matrix, where M is the number of sections in the second-order section filter. Each row of the SOS matrix contains the numerator and denominator coefficients (b_{ik} and a_{ik}) of the corresponding section in the filter.

$$\begin{bmatrix} b_{01} & b_{11} & b_{21} & a_{01} & a_{11} & a_{21} \\ b_{02} & b_{12} & b_{22} & a_{02} & a_{12} & a_{22} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{0M} & b_{1M} & b_{2M} & a_{0M} & a_{1M} & a_{2M} \end{bmatrix}$$

This parameter is only visible when **Dialog parameters** is selected.

Scale values

The **Scale values** parameter specifies the scalar or vector of M+1 scale values to be used between SOS sections.

- When you enter a scalar, the value specifies the gain value before the first section of the second-order filter. The rest of the gain values default to 1.

- When you enter a vector of M+1 values, each value specifies a separate section of the filter. For example, the first element is the first gain value, the second element is the second gain value, and so on.

  This parameter is only visible when **Dialog parameters** is selected.

Initial conditions

Specify the initial conditions of the filter states. To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-128.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is Direct form II or Direct form II transposed.

Initial conditions on zeros side

Specify the initial conditions for the filter states on the side of the filter structure with the zeros (b_0, b_1, b_2, ...); see the next diagram.

This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is Direct form I or Direct form I transposed. To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-128.

Initial conditions on poles side

Specify the initial conditions for the filter states on the side of the filter structure with the poles (a_0, a_1, a_2, ...); see the next diagram.
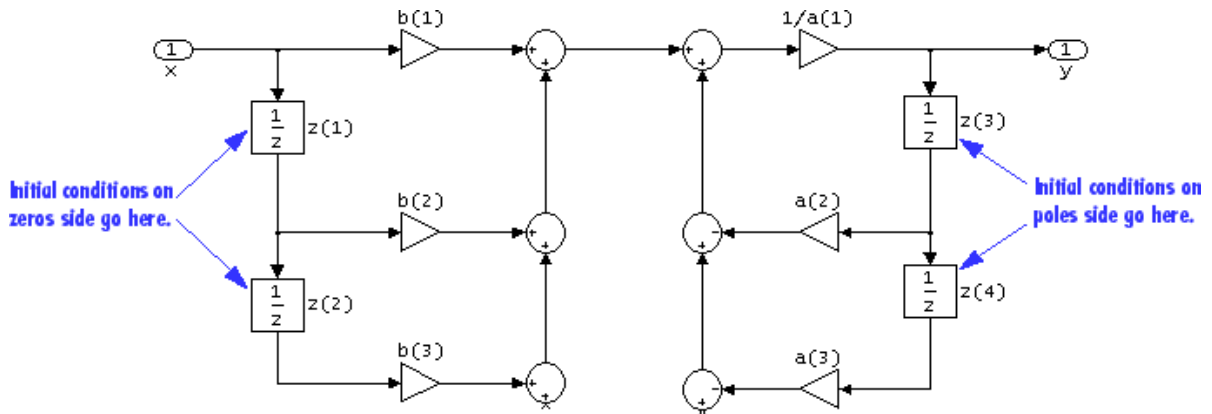
This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is Direct form I or Direct form I transposed. To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-128.

Biquad Filter

Action when the a0 values of the SOS matrix are not one
Specify the action the block should perform when the SOS matrix a_{0j} values do not equal one; None, Error, or Warn.

This parameter is only visible when **Dialog parameters** is selected.

Optimize unity scale values
Select this check box to optimize your simulation when one or more scale values equal 1. Selecting this option removes the unity gains so that the values are treated like Simulink lines or wires. In some fixed-point cases when there are unity scale values, selecting this parameter also omits certain casts. Refer to "Filter Structure Diagrams" on page 2-148 for more information.

This parameter is only visible when **Dialog parameters** is selected.

Input processing
Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Specify Filter Characteristics via Input Port

The **Main** pane of the Biquad Filter block dialog appears as follows when **Input port(s)** is selected in the **Coefficient source** group box.

Biquad Filter

Filter structure

> Select the filter structure.
>
> This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected.

Initial conditions

> Specify the initial conditions of the filter states. To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-128.
>
> This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is `Direct form II` or `Direct form II transposed`.

Initial conditions on zeros side

> Specify the initial conditions for the filter states on the side of the filter structure with the zeros (b_0, b_1, b_2, ...); see the next diagram.
>
> This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is `Direct form I` or `Direct form I transposed`. To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-128.

Initial conditions on poles side

> Specify the initial conditions for the filter states on the side of the filter structure with the poles (a_0, a_1, a_2, ...); see the next diagram.
>
> This parameter is only visible when **Dialog parameters** or **Input port(s)** is selected and the filter structure is `Direct form I` or `Direct form I transposed`. To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-128.

Biquad Filter

Scale values mode

Choose how to specify the scale values to use between filter sections. When you select `Specify via input port (g)`, you enter the scale values as a 2-D vector at port g. When you select `Assume all are unity and optimize`, all scale values are removed and treated like Simulink lines or wires.

This parameter is only visible when **Input port(s)** is selected.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The Inherited (this choice will be removed - see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Specify Discrete-Time Filter Object

The **Main** pane of the Biquad Filter block dialog appears as follows when **Discrete-time filter object (DFILT)** is selected in the **Coefficient source** group box.
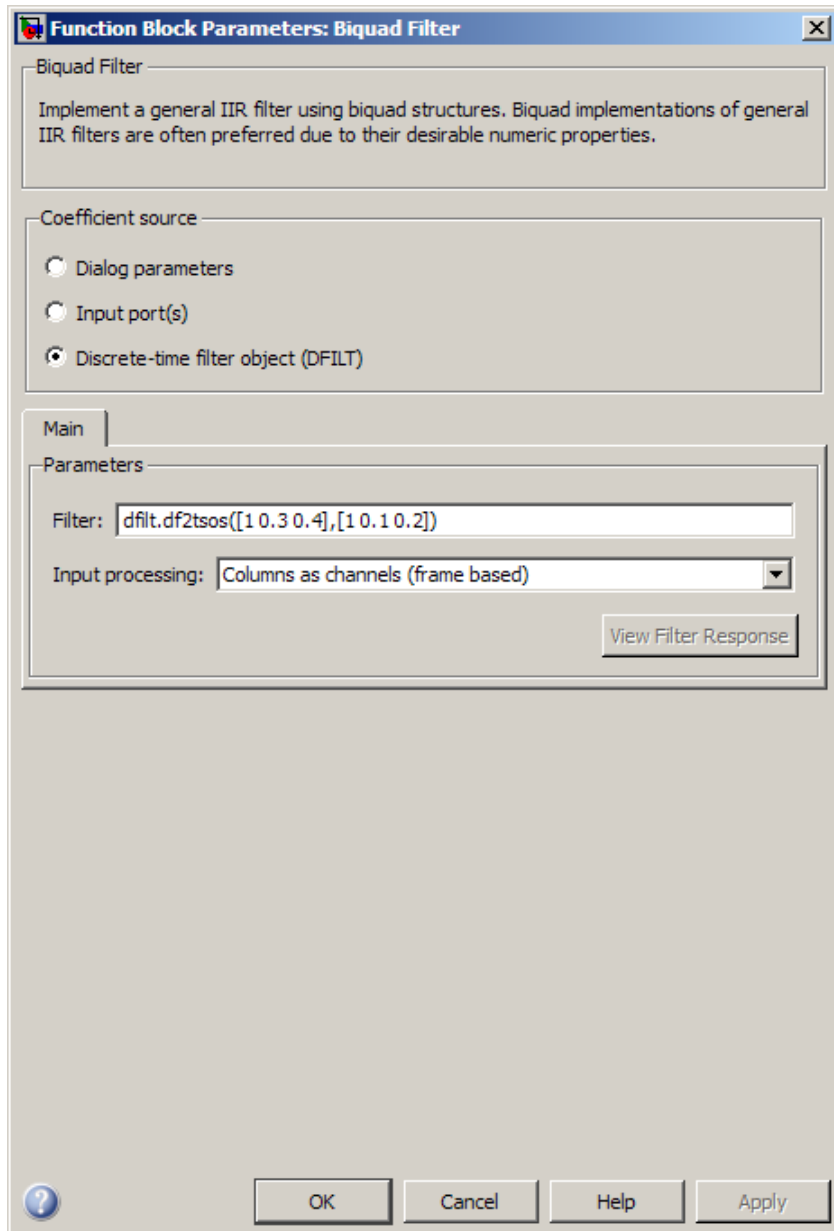
Biquad Filter

Filter

Specify the discrete-time filter object (`dfilt`) that you would like the block to implement. You can do so in one of three ways:

- You can fully specify the `dfilt` object in the block mask.

- You can enter the variable name of a `dfilt` object that is defined in any workspace.

- You can enter a variable name for a `dfilt` object that is not yet defined.

For more information on creating `dfilt` objects, see the `dfilt` reference page.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the `dfilt` object specified in the **Discrete-time filter object (DFILT)** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note If you specify a filter in the **Discrete-time filter object (DFILT)** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Biquad Filter

Specify Fixed-Point Parameters

The **Data Types** pane of the Biquad Filter block dialog appears as follows. This pane only appears when **Dialog parameters** or **Input port(s)** is selected in the **Coefficient source** group box.

Biquad Filter

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; instead, they always round to `Nearest`.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; instead, they are always saturated.

Multiplicand

Choose how you specify the word length and the fraction length of the multiplicand data type of a `Direct form I transposed` filter structure. See "Fixed-Point Data Types" on page 2-129 and the "Direct Form I Transposed" on page 2-151 filter structure diagram for illustrations depicting the use of the multiplicand data type in this block.

This parameter is only visible when the **Filter structure** parameter is set to `Direct form I transposed`.

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

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

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

Section input

Choose how you specify the word length and the fraction length of the fixed-point data type going into each section of a biquadratic filter. See "Fixed-Point Data Types" on page 2-129 for illustrations depicting the use of the section input data type in this block.

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

- When you select `Binary point scaling`, you can enter the word and fraction lengths of the section input and output, in bits.

- When you select `Slope and bias scaling`, you can enter the word lengths, in bits, and the slopes of the section input and output. This block requires power-of-two slope and a bias of zero.

Section output

Choose how you specify the word length and the fraction length of the fixed-point data type coming out of each section of a biquadratic filter. See "Fixed-Point Data Types" on page 2-129 for illustrations depicting the use of the section input data type in this block.

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

- When you select `Binary point scaling`, you can enter the word and fraction lengths of the section input and output, in bits.

- When you select `Slope and bias scaling`, you can enter the word lengths, in bits, and the slopes of the section input and output. This block requires power-of-two slope and a bias of zero.

Coefficients

Choose how you specify the word length and the fraction length of the filter coefficients (numerator, denominator, and scale value) when **Dialog parameters** is selected in the **Coefficient source** group box. See "Fixed-Point Data Types" on page 2-129 for illustrations depicting the use of the coefficient data types in this block:

- When you select `Same word length as input`, the word length of the filter coefficients match that of the input to the block. In this mode, the block automatically sets the fraction length of the coefficients to the binary-point only scaling that

provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Specify word length`, you can enter the word length of the coefficients, in bits. In this mode, the block automatically sets the fraction length of the coefficients to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Binary point scaling`, you can enter the word length and the fraction length of the coefficients, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator coefficients.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the coefficients. If applicable, you can enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.

- The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; instead, they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See "Fixed-Point Data Types" on page 2-129 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block:

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

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

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

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See "Fixed-Point Data Types" on page 2-129 and "Multiplication Data Types" for illustrations depicting the use of the accumulator data type in this block:

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

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

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

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

State

Use this parameter to specify how you would like to designate the state word and fraction lengths when **Dialog parameters** is selected in the **Coefficient source** group box. See "Fixed-Point Data Types" on page 2-129 for illustrations depicting the use of the state data type in this block.

This parameter is not visible for `Direct form I` and `Direct form I transposed filter` structures.

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

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

Biquad Filter

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

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

Output

Choose how you specify the output word length and fraction length. See "Fixed-Point Data Types" on page 2-129 for illustrations depicting the use of the output data type in this block.

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

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

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

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

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Filter Structure Diagrams

The diagrams in the following sections show the filter structures supported by the Biquad Filter block. They also show the data types used in the filter structures for fixed-point signals. You can set the data types shown in these diagrams in the block dialog box. This is discussed in "Dialog Box" on page 2-129.

- "Direct Form I" on page 2-149
- "Direct Form I Transposed" on page 2-151
- "Direct Form II" on page 2-154
- "Direct Form II Transposed" on page 2-157

Direct Form I
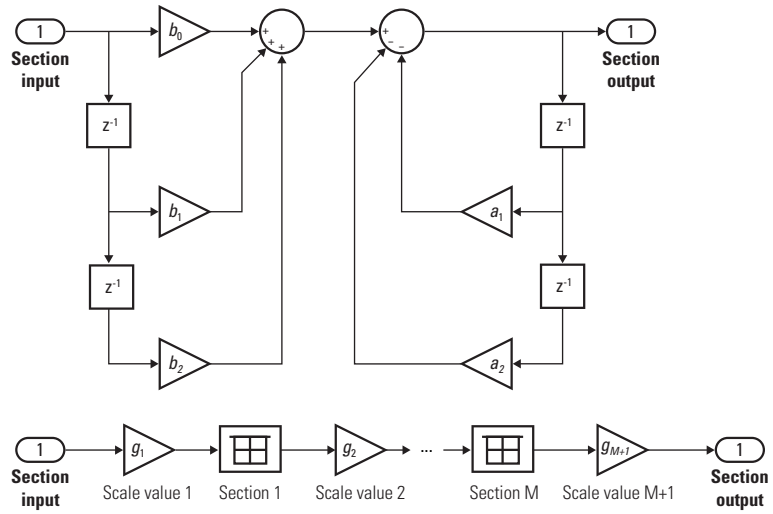
Biquad Filter

The following diagram shows the data types for one section of the filter for fixed-point signals.
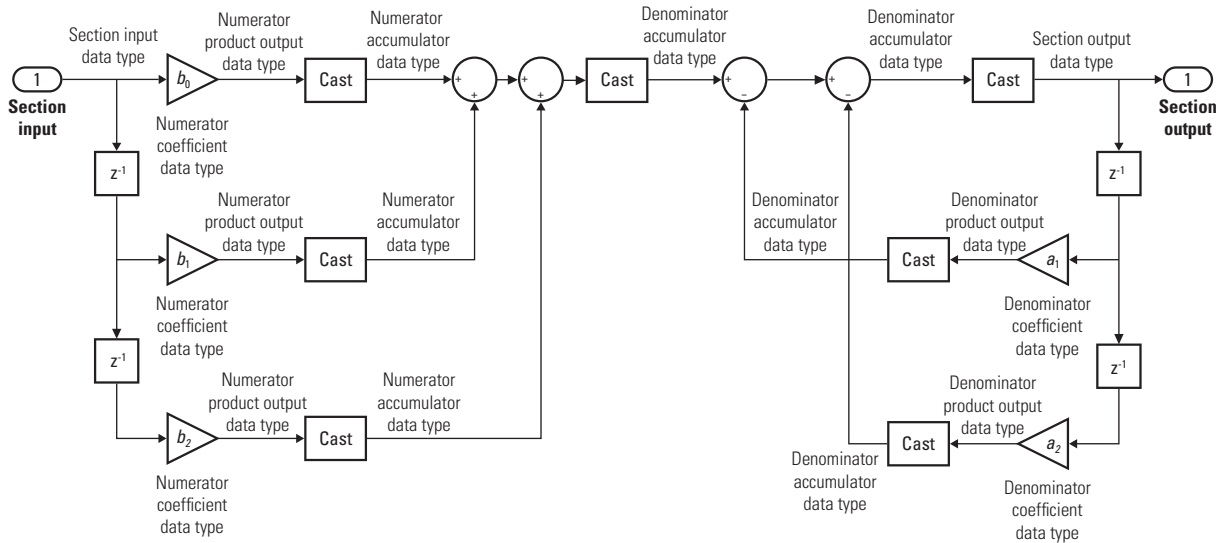
The following diagrams show the fixed-point data types between filter sections.

When the data is not optimized:
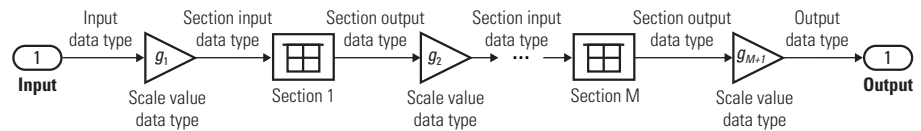
When you select **Optimize unity scale values** and scale values equal 1:
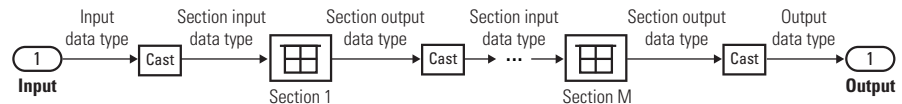
Direct Form I Transposed
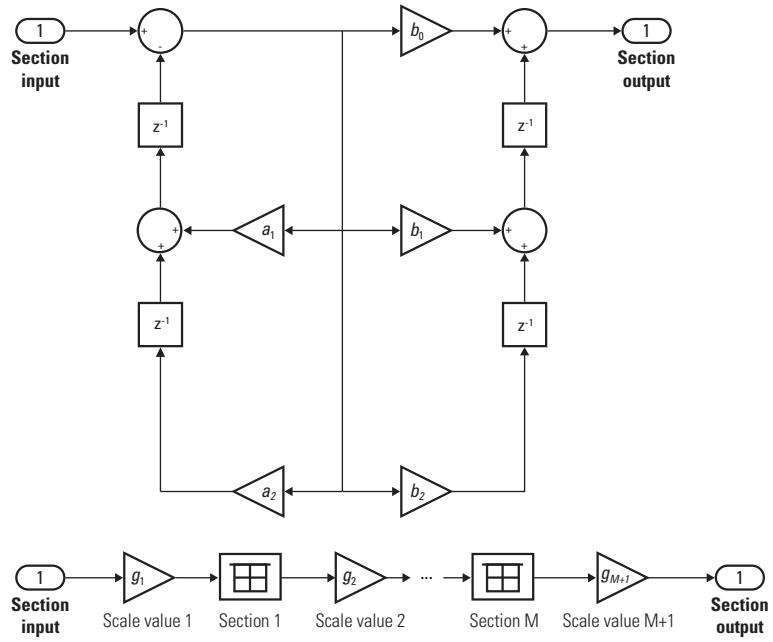
Biquad Filter

The following diagram shows the data types for one section of the filter for fixed-point signals.
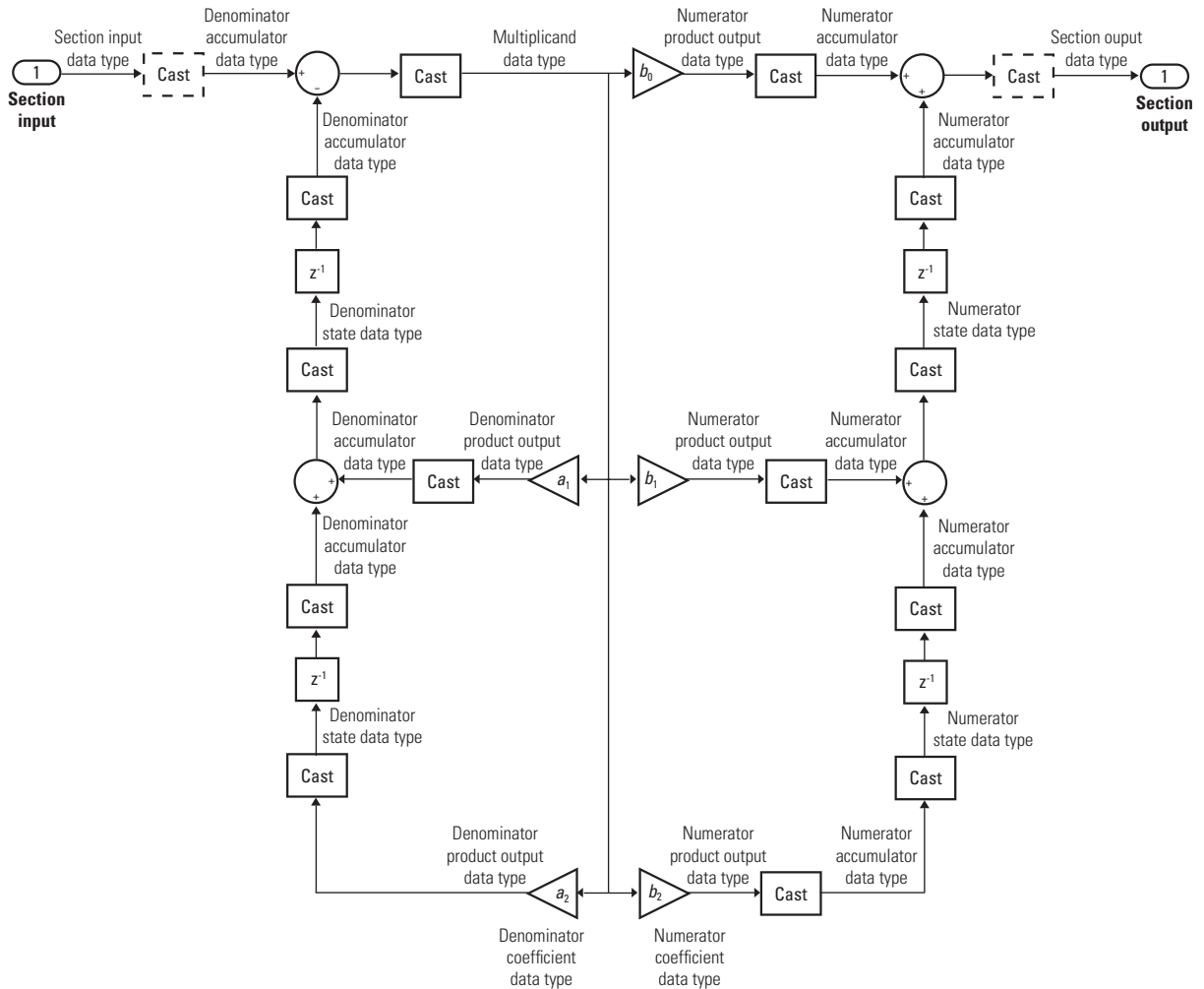
The dashed casts are omitted when **Optimize unity scale values** is selected and scale values equal one.

The following diagrams show the fixed-point data types between filter sections.

When the data is not optimized:
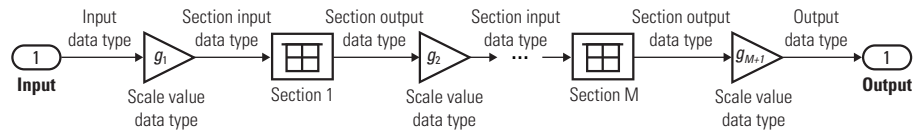
When you select **Optimize unity scale values** and scale values equal 1:
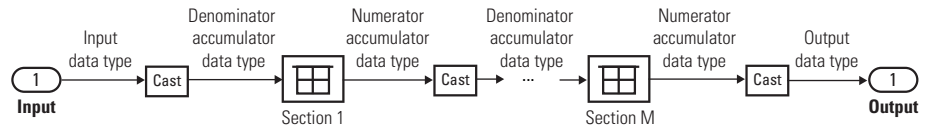
Biquad Filter

Direct Form II
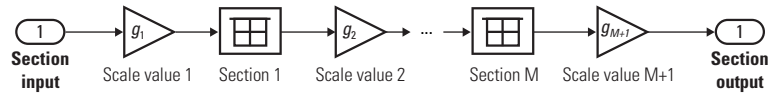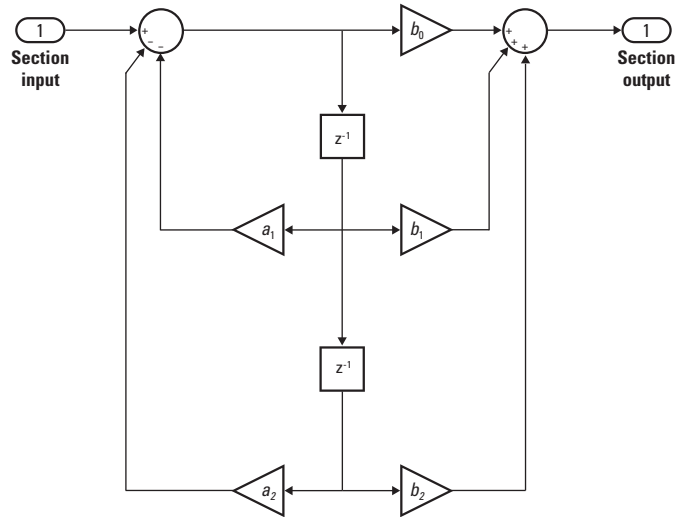
The following diagram shows the data types for one section of the filter for fixed-point signals.
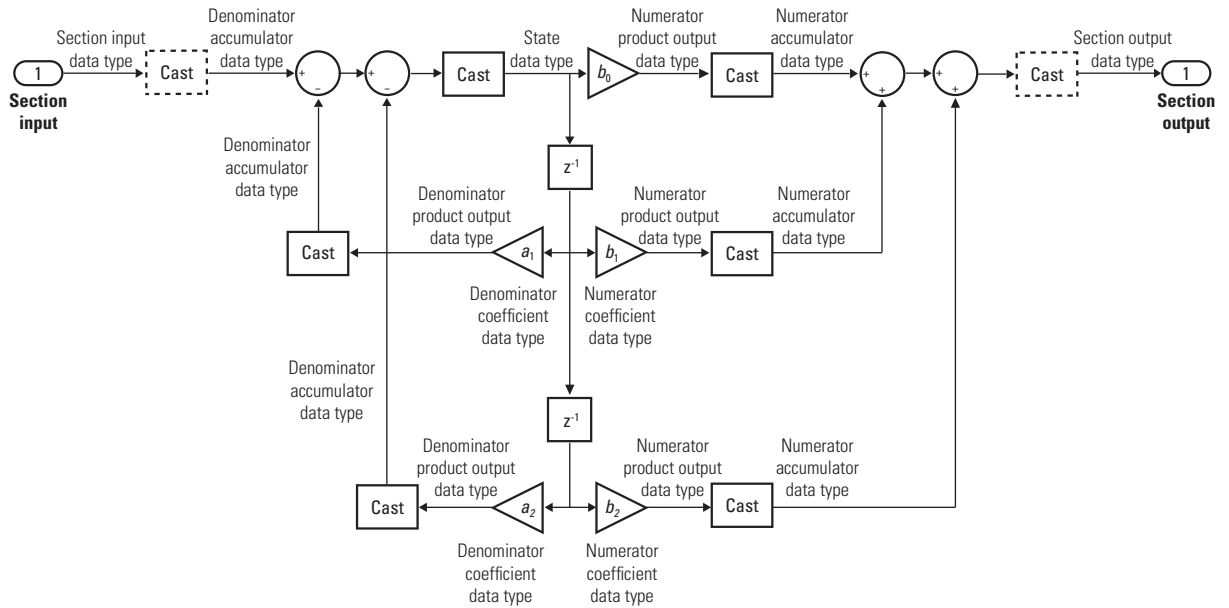
The dashed casts are omitted when **Optimize unity scale values** is selected and scale values equal one.

Biquad Filter

The following diagrams show the fixed-point data types between filter sections.

When the data is not optimized:
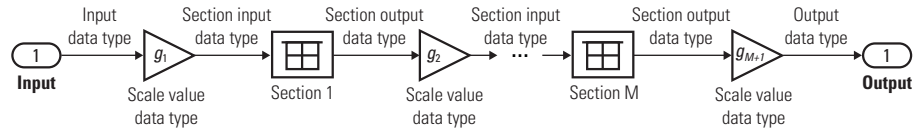
When you select **Optimize unity scale values** and scale values equal 1:
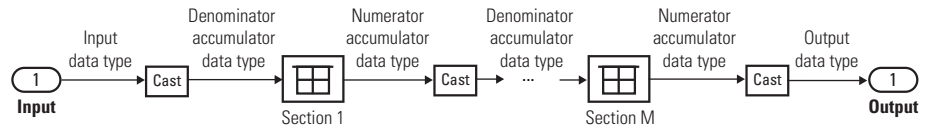
Direct Form II Transposed
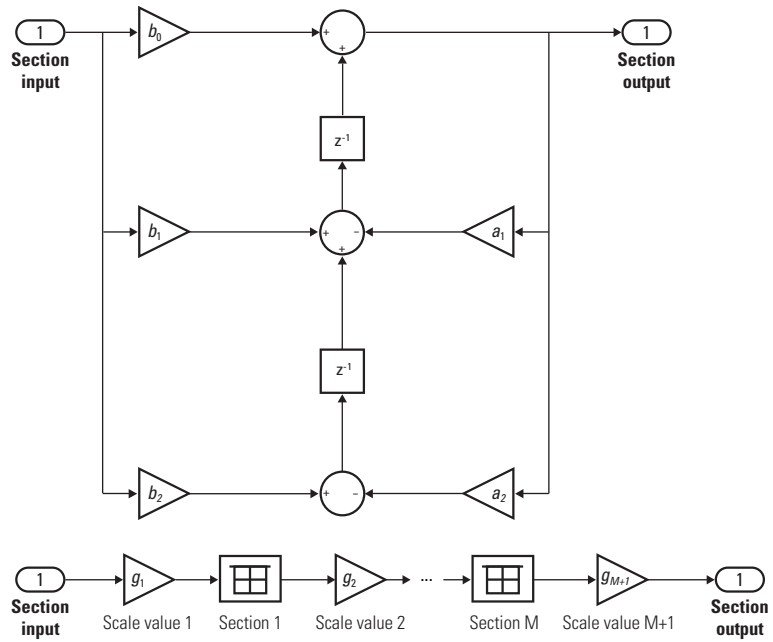
Biquad Filter

The following diagram shows the data types for one section of the filter for fixed-point signals.
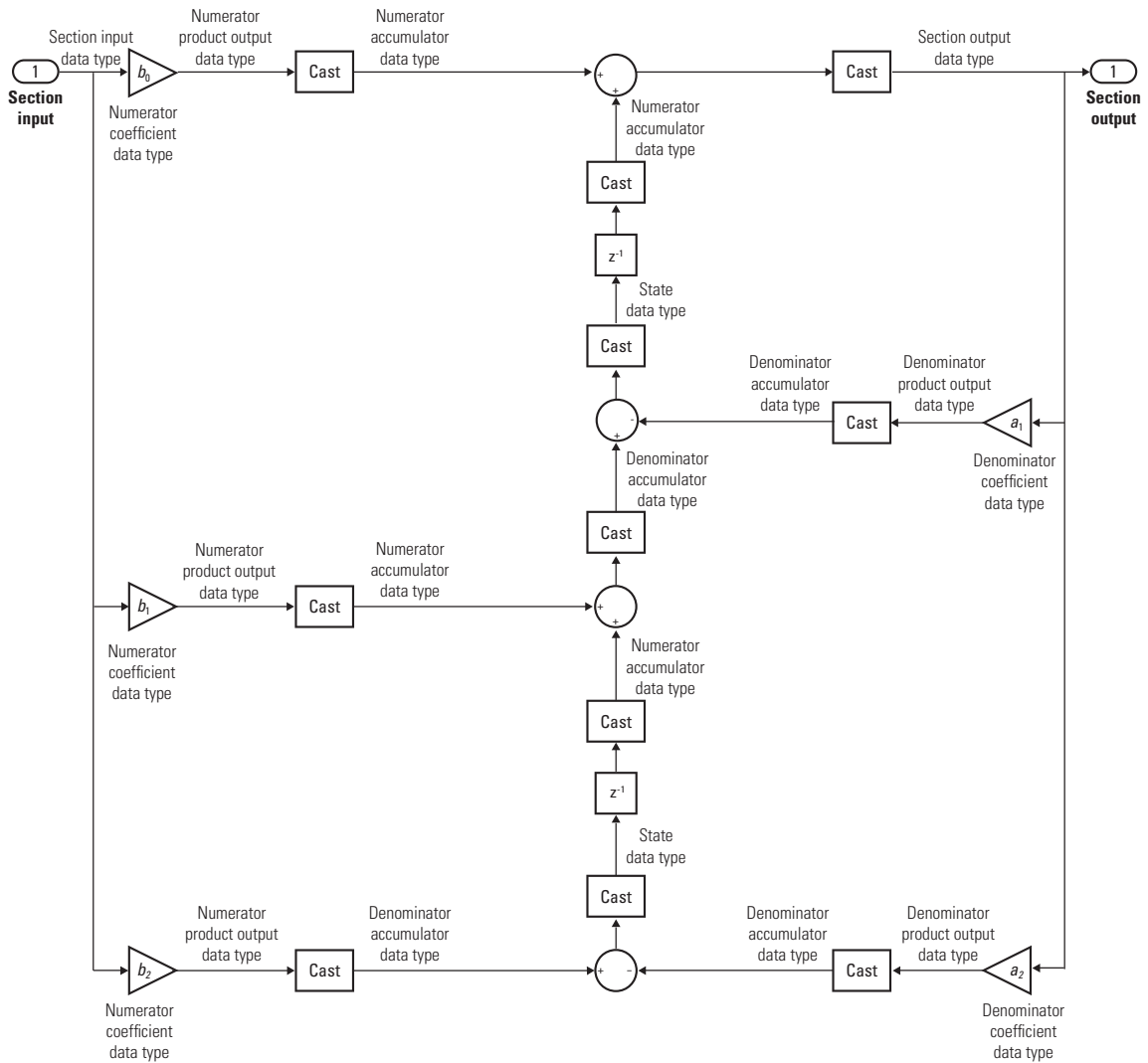
The following diagrams show the fixed-point data types between filter sections.

When the data is not optimized:
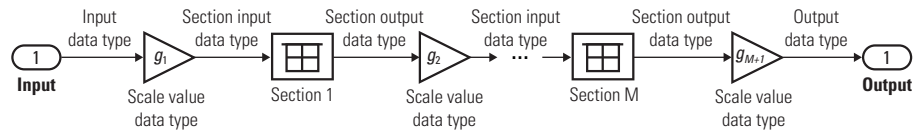
When you select **Optimize unity scale values** and scale values equal 1:

<table>
<tr><td rowspan="3">**Supported Data Types**</td></tr>
</table>

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

See Also

dfilt.df1sos   DSP System Toolbox

dfilt.df1tsos   DSP System Toolbox

Biquad Filter

dfilt.df2sos   DSP System Toolbox

dfilt.df2tsos   DSP System Toolbox

Purpose   Compute output, error, and weights using LMS adaptive algorithm

Library   Filtering / Adaptive Filters

dspadpt3

Description

Block diagram:
```
Input        Output
Desired  Block
Step-size LMS  Error
Adapt
Reset         Wts
```
Block LMS Filter
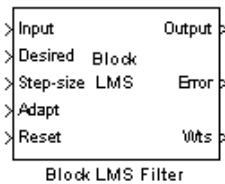
The Block LMS Filter block implements an adaptive least mean-square (LMS) filter, where the adaptation of filter weights occurs once for every block of samples. The block estimates the filter weights, or coefficients, needed to minimize the error, $e(n)$, between the output signal, $y(n)$, and the desired signal, $d(n)$. Connect the signal you want to filter to the Input port. The input signal can be a scalar or a column vector. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal. The Error port outputs the result of subtracting the output signal from the desired signal.

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

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

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

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

The variables are as follows.

Block LMS Filter

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

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

The **Block size** parameter determines how many samples of the input signal are acquired before the filter weights are updated. The number of rows in the input must be an integer multiple of the **Block size** parameter.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Dialog Box

Filter length

Enter the length of the FIR filter weights vector.

Block size

Enter the number of samples to acquire before the filter weights are updated. The number of rows in the input must be an integer multiple of the **Block size**.

Specify step-size via

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

Step-size (mu)

Enter the step-size. Tunable.

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \le 1$. Tunable.

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Adapt port

Select this check box to enable the Adapt input port.

Reset port

Select this check box to enable the Reset input port.

Output filter weights

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

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

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Desired	• Must be the same as Input
Step-size	• Must be the same as Input

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

See Also

Fast Block LMS Filter	DSP System Toolbox
Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
LMS Filter	DSP System Toolbox
RLS Filter	DSP System Toolbox

See "Adaptive Filters in Simulink" for related information.

Buffer

Purpose   Buffer input sequence to smaller or larger frame size

Library   Signal Management / Buffers

dspbuff3
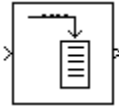
Description The Buffer block always performs frame-based processing. The block
redistributes the data in each column of the input to produce an output
with a different frame size. Buffering a signal to a larger frame size
yields an output with a *slower* frame rate than the input. For example,
consider the following illustration for scalar input.
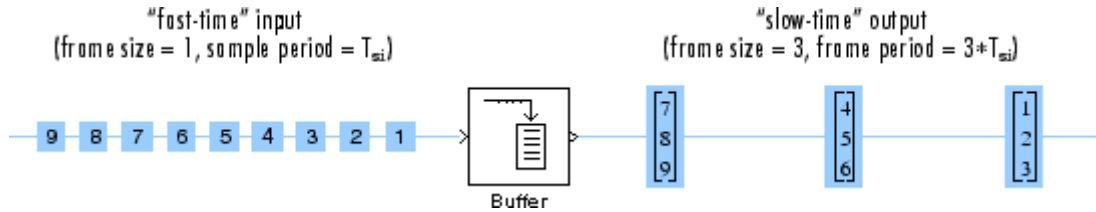
Buffering a signal to a smaller frame size yields an output with a
faster frame rate than the input. For example, consider the following
illustration of scalar output.
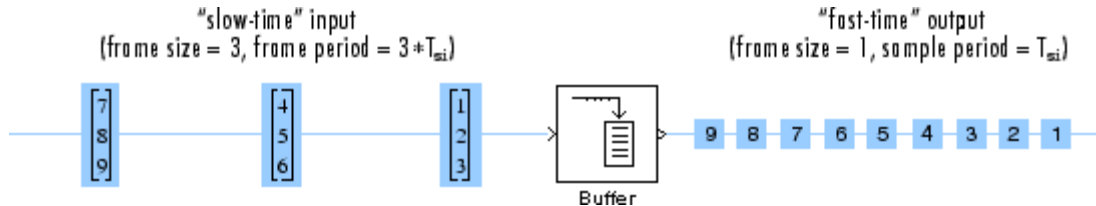
The block coordinates the output *frame size* and *frame rate* of
nonoverlapping buffers such that the sample period of the signal is the
same at both the input and output: $T_{so} = T_{si}$.

This block supports triggered subsystems when the block input and
output rates are the same.

Buffering Single Channel Signals

The following table shows the output dimensions of the Buffer block when the input is a single-channel signal. M_o is the value of the **Output buffer size** parameter.

Input Dimensions	Output Dimensions
1-by-1 (scalar)	M_o-by-1
M_i-by-1 (column vector)	M_o-by-1

The input frame period is $M_i \cdot T_{si}$, where M_i is the input frame size and

T_{si} is the input sample period. The output frame period is $(M_o - L)T_{si}$, where L is the value of the **Buffer overlap** parameter and T_{si} is the input sample period. When you set the **Buffer overlap** parameter to $M_o - 1$, the output frame period equals the input sample period.

Buffering Multichannel Signals

The following table shows the output dimensions of the Buffer block when the input is a multichannel signal. M_o is the value of the **Output buffer size** parameter and can be greater or less than the input frame size, M_i. The block buffers each of the N input channels independently.

Input Dimensions	Output Dimensions
1-by-N (row vector)	M_o-by-N
M_i-by-N (matrix)	M_o-by-N

The input frame period is $M_i \cdot T_{si}$, where M_i is the input frame size and

T_{si} is the input sample period. The output frame period is $(M_o - L)T_{si}$, which equals the sequence sample period when the **Buffer overlap** is

$M_o - 1$. Thus, the output sample period is related to the input sample period by

Buffer

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

Buffering with Overlap or Underlap

The **Buffer overlap** parameter, *L*, specifies the amount of overlap or underlap in each successive output frame. To overlap the data in the buffer, specify a value of *L* in the range $0 \le L < M_o$, where M_o is the value of the **Output buffer size** parameter. The block takes *L* samples (rows) from the current output and repeats them in the next output. In cases of overlap, the block acquires $M_o - L$ *new* input samples before propagating the buffered data to the output.

When $L < 0$, you are buffering the signal with underlap. The block discards *L* input samples after the buffer fills and outputs the buffer with period $(M_o - L)T_{si}$, which is longer than in the zero-overlap case.

The output frame period is $(M_o - L)T_{si}$, which equals the input sequence sample period, T_{si}, when the **Buffer overlap** is $M_o - 1$.

Latency

Zero-tasking latency means that the first input sample, received at *t* = 0, appears as the first output sample. In the Simulink single-tasking mode, the Buffer block has zero-tasking latency for the following special cases:

- Scalar input and output ($M_o = M_i = 1$) with zero or negative **Buffer overlap** ($L \le 0$)

- Input frame size is an integer multiple of the output frame size

  $$M_i = kM_o$$

  where *k* is an integer with zero **Buffer overlap** ($L = 0$); notable cases of this include

- Any input frame size M_i with scalar output ($M_o = 1$) and zero **Buffer overlap** ($L = 0$)

- Equal input and output frame sizes ($M_o = M_i$) with zero **Buffer overlap** ($L = 0$)

For all cases of 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. The block reads from this buffer to generate the first D output samples, where

$$D = \begin{cases} M_o + L & (L \geq 0) \\ M_o & (L < 0) \end{cases}$$

The dimensions of the **Initial conditions** parameter depend on the **Buffer overlap**, L, and whether the input is single-channel or multichannel:

- When $L \neq 0$, the **Initial conditions** parameter must be a scalar.

- When $L = 0$, the **Initial conditions** parameter can be a scalar, or it can be a vector with the following constraints:

  - For single-channel inputs, the **Initial conditions** parameter can be a vector of length M_o if M_i is 1, or a vector of length M_i if M_o is 1.

  - For multichannel inputs, the **Initial conditions** parameter can be a vector of length $M_o * N$ if M_i is 1, or a vector of length $M_i * N$ if M_o is 1.

For all *multitasking* operations, 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 a 1-by-N row vector, and the output of the block is 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 a 1-by-N row vector, which is the result of unbuffering an M_i-by-N matrix, the **Initial conditions** can be

- A vector containing M_i samples to output sequentially for each channel during the first M_i sample times
- A scalar to be repeated across all elements of the initial output(s)

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder™ User's Guide*.

Behavior in Enabled Subsystems

The Buffer block cannot be used in an enabled subsystem under the following conditions:

- In a multirate multitasking environment
- When the **Buffer overlap** parameter is set to a negative value

The Buffer block has an internal reservoir that temporarily stores data. When the Buffer block is used in an enabled subsystem, there is the possibility that the reservoir can overrun or underrun. The block implements safeguards against these occurrences.

Overrun occurs when more data enters into the buffer than it can hold. For example, consider buffering a scalar input to a frame of size three

with a buffer that accepts input every second and outputs every three seconds. If you place this buffer inside an enabled subsystem that is disabled every three seconds at $t = 3$s, $t = 6$s, and so on, the buffer accumulates data in its internal reservoir without being able to empty it. This condition results in overrun.

Underrun occurs when the buffer runs out of data to output. For example, again consider buffering a scalar input to a frame size of three with a buffer that accepts input every second and outputs every three seconds. If you place this buffer inside an enabled subsystem that is disabled at $t = 10$s, $t = 11$s, $t = 13$s, $t = 14$s, $t = 16$s, and $t = 17$s, its internal reservoir becomes drained, and there is no data to output at $t = 18$s. This condition results in underrun.

To protect from overrun and underrun, the Buffer block keeps a record of the amount of data in its internal reservoir. When the Buffer block reads data, the amount of data in its reservoir goes up. When data is output from the Buffer block, the amount of data in its reservoir goes down. To protect from overrun, the oldest samples in the reservoir are discarded whenever amount of data in the reservoir is larger than the actual buffer size. To protect from underrun, the most recent samples are repeated whenever an output is due and there is no data in the reservoir.

Examples   **Buffering a two-channel input with overlap**

The ex_buffer_tut4 model buffers a two-channel input using an **Output buffer size** of three and a **Buffer overlap** of one. The following diagram illustrates the inputs and outputs of the Buffer block.

Buffer
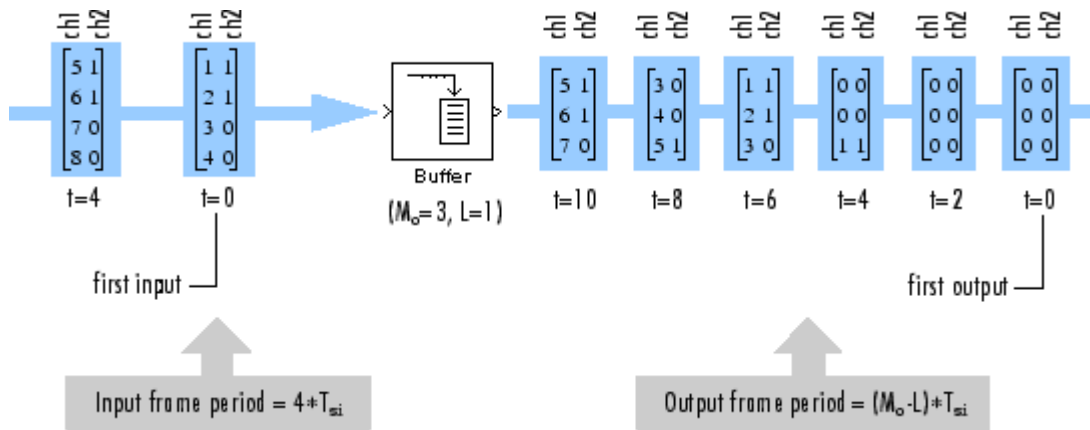
Notice that the output is delayed by eight samples. This is latency occurs because of the parameter settings in this model, and because the model is running in Simulink multitasking mode. The first eight output samples therefore adopt the value specified for the **Initial conditions** parameter, which in this case is set to zero. You can use the rebuffer_delay function to determine the latency of the Buffer block for any combination of frame size and overlap values.

Buffering a four-channel input with overlap

The ex_buffer_tut3 buffers a four-channel input using a **Output buffer size** of three and a **Buffer overlap** of one. The following diagram illustrates the inputs and outputs of the Buffer block.
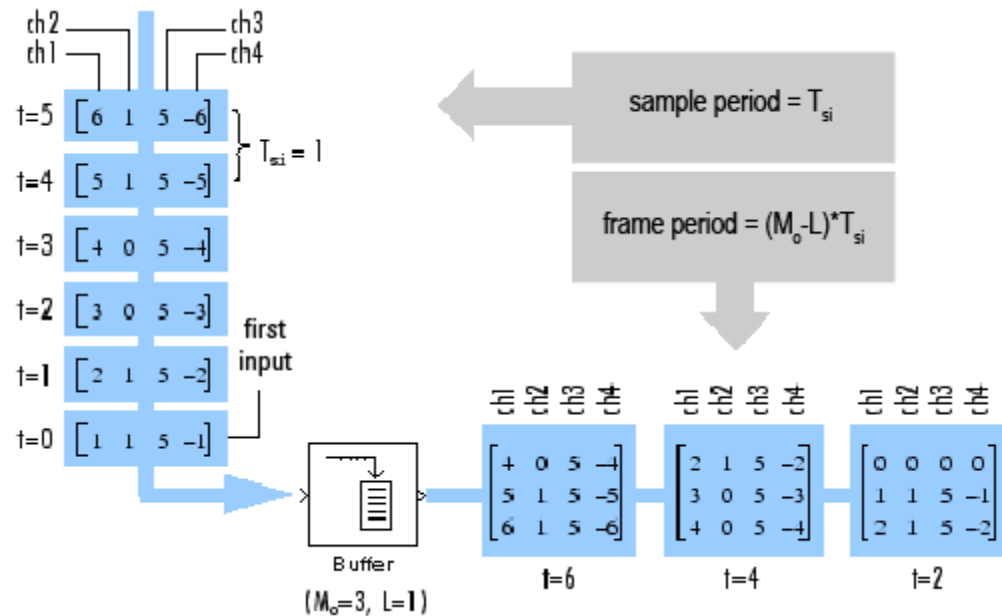
Notice that the input vectors do not begin appearing at the output until the second row of the second matrix. This is due to latency in the Buffer block. The first output matrix (all zeros in this example) reflects the value of the **Initial conditions** parameter, while the first row of zeros in the 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 the previous example,

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

This agrees with the four samples of delay (zeros) per channel shown in the previous figure.

Buffer
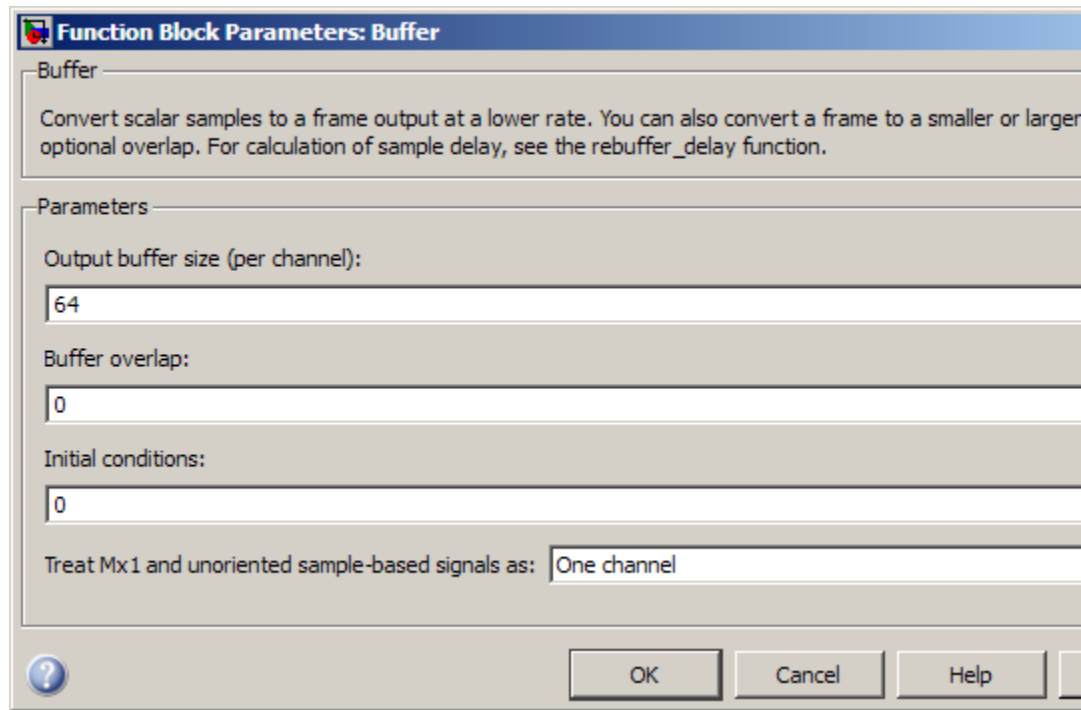
Dialog Box

Output buffer size

Specify the number of consecutive samples, M_o, from each channel to buffer into the output frame.

Buffer overlap

Specify the number of samples, L, by which consecutive output frames overlap.

Initial conditions

Specify the value of the block's initial output for cases of nonzero latency; a scalar, vector, or matrix.

Treat Mx1 and unoriented sample-based signals as
Specify how the block treats sample-based M-by-1 column vectors and unoriented sample-based vectors of length M. You can select one of the following options:

- One channel — When you select this option, the block treats M-by-1 and unoriented sample-based inputs as a column vector (one channel). In this mode, the output of the block is an M_o-by-1 column vector, where M_o is the **Output buffer size**.

- M channels (this choice will be removed   see release notes) — When you select this option, the block treats M-by-1 and unoriented sample-based inputs as a 1-by-M row vector. Because the block always does frame-based processing, the block interprets the 1-by-M row vector as M individual channels. In this mode, the output of the block is an M_o-by-M matrix, where M_o is the **Output buffer size**, and M is the length of the sample-based input vector.

Note This parameter will be removed in a future release. At that time, the Buffer block will always perform frame-based processing.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• Boolean
	• 8-, 16-, and 32-bit signed integers

Buffer

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

See Also

Delay Line	DSP System Toolbox
Unbuffer	DSP System Toolbox
rebuffer_delay	DSP System Toolbox

See "Convert Sample and Frame Rates in Simulink" and "Buffering and Frame-Based Processing" for more information.

Purpose   Compute estimate of autoregressive (AR) model parameters using Burg method

Library   Estimation / Parametric Estimation

dspparest3
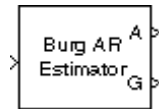
Description   The Burg AR Estimator block uses the Burg method to fit an autoregressive (AR) model to the input data by minimizing (least squares) the forward and backward prediction errors while constraining the AR parameters to satisfy the Levinson-Durbin recursion.

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 you select the **Inherit estimation order from input dimensions** parameter, the order, p, of the all-pole model is one less that the length of the input vector. Otherwise, the order is the value specified by the **Estimation order** parameter.

The **Output(s)** parameter allows you to select between two realizations of the AR process:

- A — The top output, A, is a column vector of length $p+1$ with the same frame status as the input, and contains the normalized estimate of the AR model polynomial coefficients in descending powers of z.

  [1 a(2) ... a(p+1)]

Burg AR Estimator

- 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).

The following table compares the features of the Burg AR Estimator block to the Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

	Burg AR Estimator	**Covariance AR Estimator**	**Modified Covariance AR Estimator**	**Yule-Walker AR Estimator**
Characteristics	Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
	Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion	Minimizes the forward prediction error in the least squares sense	Minimizes the forward and backward prediction errors in the least squares sense	Minimizes the forward prediction error in the least squares sense (also called "autocorrelation method")
Advantages	Always produces a stable model			Always produces a stable model

	Burg AR Estimator	Covariance AR Estimator	Modified Covariance AR Estimator	Yule-Walker AR Estimator
Disadvantages		May produce unstable models	May produce unstable models	Performs relatively poorly for short data records
Conditions for Nonsingularity		Order must be less than or equal to half the input frame size	Order must be less than or equal to 2/3 the input frame size	Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular

Burg AR Estimator
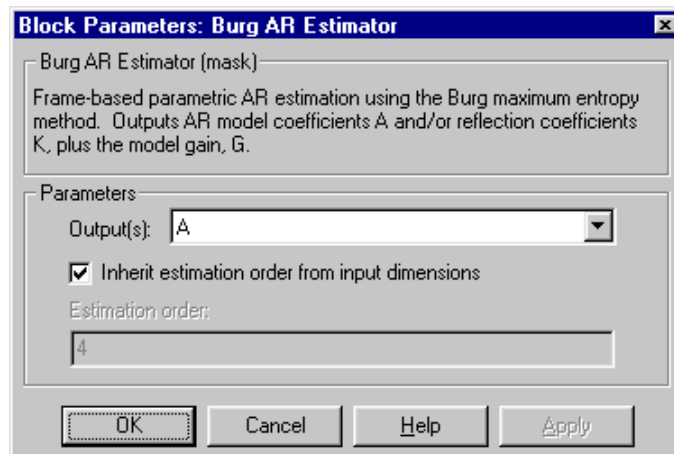
Dialog Box

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 you do not select **Inherit estimation order from input dimensions**.

References Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
A	• Double-precision floating point
	• Single-precision floating point
G	• Double-precision floating point
	• Single-precision floating point

See Also

Burg Method	DSP System Toolbox
Covariance AR Estimator	DSP System Toolbox
Modified Covariance AR Estimator	DSP System Toolbox
Yule-Walker AR Estimator	DSP System Toolbox
arburg	Signal Processing Toolbox

Burg Method

Purpose   Power spectral density estimate using Burg method

Library   Estimation / Power Spectrum Estimation

dspspect3

Description
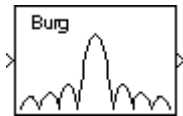
The Burg Method block estimates the power spectral density (PSD) of the input frame using the Burg method. This method fits an autoregressive (AR) model to the signal by minimizing (least squares) the forward and backward prediction errors. Such minimization occurs with the AR parameters constrained to satisfy the Levinson-Durbin recursion.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only). This input represents a frame of consecutive time samples from a single-channel signal. The block outputs a column vector containing the estimate of the power spectral density of the signal at N_{fft} equally spaced frequency points. The frequency points are in the range $[0,F_s)$, where F_s is the sampling frequency of the signal.

When you select the **Inherit estimation order from input dimensions** parameter, the order of the all-pole model is one less than the input frame size. Otherwise, the **Estimation order** parameter specifies the order. The block computes the spectrum from the FFT of the estimated AR model parameters.

Selecting the **Inherit FFT length from estimation order** parameter specifies that N_{fft} is one greater than the estimation order. Clearing the **Inherit FFT length from estimation order** check box, allows you to use the **FFT length** parameter to specify N_{fft} as a power of 2. The block zero-pads or wraps the input to N_{fft} before computing the FFT. The output is always sample based.

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must 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 equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

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*)

Burg Method

	Burg	Covariance	Modified Covariance	Yule-Walker
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

	Burg	Covariance	Modified Covariance	Yule-Walker
	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 be positive-definite, hence nonsingular

Examples   The dspsacomp example compares the Burg method with several other spectral estimation methods.

Burg Method
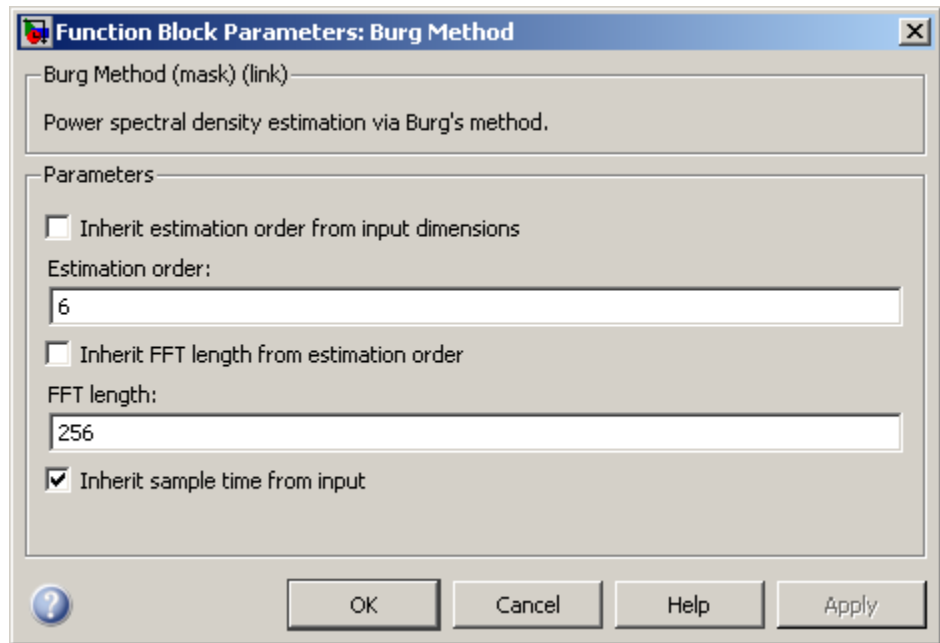
Inherit estimation order from input dimensions

Selecting this check box sets the estimation order to one less than the length of the input vector.

Estimation order

The order of the AR model. This parameter becomes visible only when you clear the **Inherit estimation order from input dimensions** check box.

Inherit FFT length from estimation order

When selected, the FFT length is one greater than the estimation order. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from estimation order** check box. You can then specify a power-of-two FFT length using the **FFT length** parameter.

FFT length

Enter the number of data points on which to perform the FFT, N_{fft}. When N_{fft} is larger than the input frame size, the block zero-pads each frame as needed. When N_{fft} is smaller than the input frame size, the block wraps each frame as needed. This parameter becomes visible only when you clear the **Inherit FFT length from input dimensions** check box.

Inherit sample time from input

If you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must 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 equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

Sample time of original time series

Specify the sample time of the original time-domain signal. This parameter becomes visible only when you clear the **Inherit sample time from input** check box.

References   Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Orfanidis, S. J. *Introduction to Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1995.

Orfanidis, S. J. *Optimum Signal Processing: An Introduction.* 2nd ed. New York, NY: Macmillan, 1985.

Burg Method

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

See Also

Burg AR Estimator	DSP System Toolbox
Covariance Method	DSP System Toolbox
Modified Covariance Method	DSP System Toolbox
Short-Time FFT	DSP System Toolbox
Yule-Walker Method	DSP System Toolbox
spectrum.burg	Signal Processing Toolbox

See "Spectral Analysis" for related information.

Purpose

Error when input signal does or does not match selected attributes exactly

Library

Signal Management / Signal Attributes

dspsigattribs

Description
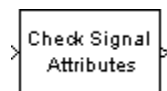
The Check Signal Attributes block terminates the simulation with an error when the input characteristics differ from the characteristics you specify in the block parameters.

When you set **Error when input** to Does not match attributes exactly, the block generates an error when the input fails to match *any* of the specified attributes. Only signals that possess *all* of the specified attributes propagate to the output unaltered and do not cause the block to generate an error.

When you set **Error when input** to Matches attributes exactly, the block generates an error only when the input possesses *all* specified attributes. Signals that do not possess *all* of the specified attributes propagate to the output unaltered, and do not cause the block to generate an error.

Signal Attributes

The Check Signal Attributes block can test for up to five different signal attributes, as specified by the following parameters. When you select Ignore for any parameter, the block does not check the signal for the corresponding attribute. For example, when you set **Complexity** to Ignore, neither real nor complex inputs cause the block to generate an error. The attributes are:

- **Complexity**

  Check whether the input is real or complex. You can display this information in a model by attaching a Probe block with **Probe complex signal** selected. Alternatively, you can select **Port Data Types** from the **Format** > **Port/Signal Displays** menu.

- **Frame status**

Check Signal Attributes

Check whether the signal is frame based or sample based. The Simulink environment displays sample-based signals using a single line and frame-based signals using a double line.

- **Dimensionality**

  Check the dimensionality of the input for compliance or noncompliance with the attributes in the subordinate **Dimension** menu. See the following table. M and N are positive integers unless otherwise indicated.

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	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)

When you select **Signal Dimensions** from the **Format > Port/Signal Displays** 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. You can also display dimension information for a signal in a model by attaching a Probe block with **Probe signal dimensions** selected.

- **Data type**

  Check the signal data type for compliance (Is...) or noncompliance (Is not...) with the attributes in the subordinate **General data type** menu. See the following table. You can individually select any of the specific data types listed in the Is... column from the subordinate **Specific data type** menu.

Check Signal Attributes

General Data Type	Is...	Is not...
Boolean	`boolean`	`single`, `double`, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`, fixed point, enumerated
Enumerated	A user-defined enumerated data type. See "Data Types" in the Simulink documentation.	`boolean`, `single`, `double`, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`, fixed point
Floating point	`single`, `double`	`boolean`, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`, fixed point, enumerated
Floating point or Boolean	`single`, `double`, `boolean`	`uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`, fixed point, enumerated
Fixed point	fixed point, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`	`boolean`, `single`, `double`, enumerated
Integer	Signed integer `int8`, `int16`, `int32` Unsigned integer `uint8`, `uint16`, `uint32`	`boolean`, `single`, `double`, fixed point, enumerated

To display data type information, in your model window, from the **Format** menu, point to **Port/Signal Displays** and select **Port Data Types**.

- **Sample time**

  Check whether the signal is discrete time or continuous time. When you select **Colors** from the **Format > Sample Time Display** menu, Simulink displays continuous-time signal lines in black or grey and discrete-time signal lines in colors corresponding to the relative rate.

  When you attach a Probe block with **Probe sample time** enabled to a continuous-time signal, the block icon displays the string `Ts:[0 To]`, where `To` is the sample time offset. Valid values of `To` for continuous-time signals are `0` and `1`. When `To` is `0`, updates occur at every major and minor time step. When `To` is `1`, updates occur only at major time steps and the sample time is *fixed in minor time step*.

  When you attach a Probe block with **Probe sample time** enabled to a discrete-time signal, the block icon displays the string `Ts:[Ts To]` for sample-based signals, and `Tf:[Tf To]` for frame-based signals. `Ts` and `Tf` are the positive sample period and frame period, respectively. `To` is the offset, such that $0 \leq offset < period$. Frame-based signals are almost always discrete time.

Check Signal Attributes
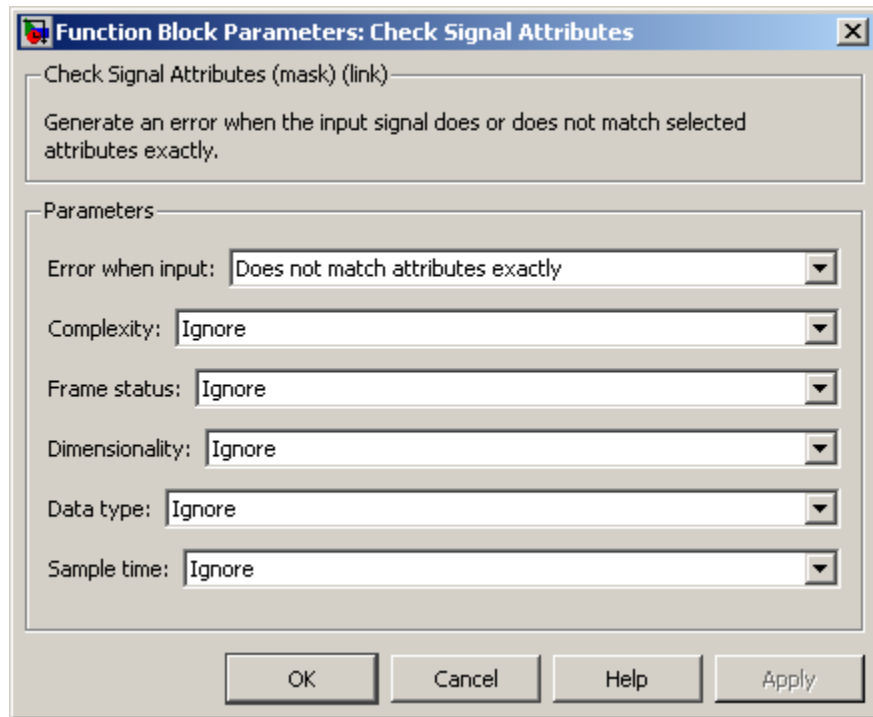
Dialog Box

Error when input

Specify whether the block generates an error when the input *does* or *does not* possess *all* of the required attributes.

Complexity

Specify the complexity for which you want to check the input, `Real` or `Complex`. When you select `Ignore` from the list, the block does not check the complexity of the input.

Frame status

Specify the frame status for which you want to check the input, `Sample-based` or `Frame-based`. When you select `Ignore` from the list, the block does not check the frame status of the input.

Dimensionality

Specify whether you want to check the input for compliance or noncompliance with the attributes in the subordinate **Dimensions** menu. When you select `Ignore` from the list, the block does not check the dimensionality of the input.

Dimensions

Specify the dimensions for which you want to check the input. This parameter is only visible when you set the **Dimensionality** parameter to `Is...` or `Is not...`.

Data type

Specifies whether you want to check the input for compliance or noncompliance with the attributes in the subordinate **General data type** menu. When you select `Ignore` from the list, the block does not check the input data type.

General data type

Specify the general data type for which you want to check the input. This parameter is only visible when you set the **Data type** to `Is...` or `Is not...`.

Specific floating-point

Specify the floating-point data type for which you want to check the input. This parameter is only visible when you set the **General data type** to `Floating-point` or `Floating-point or boolean`.

Specific fixed-point

Specify the fixed-point data type for which you want to check the input. This parameter is only visible when you set the **General data type** to `Fixed-point`.

Specific integer

Specify the integer data type for which you want to check the input. This parameter is only visible when you set the **General data type** to `Integer`.

Check Signal Attributes

Sample time
> Specify the sample time for which you want to check the input, `Discrete` or `Continuous`. When you select `Ignore` from the list, the block does not check the sample time of the input.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8, 16, and 32-bit signed integers
	• 8, 16, and 32-bit unsigned integers
	• Enumerated
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8, 16, and 32-bit signed integers
	• 8, 16, and 32-bit unsigned integers
	• Enumerated

See Also

Buffer	DSP System Toolbox
Convert 1-D to 2-D	DSP System Toolbox
Convert 2-D to 1-D	DSP System Toolbox
Data Type Conversion	Simulink

Frame Status Conversion (Obsolete)	DSP System Toolbox
Inherit Complexity	DSP System Toolbox
Probe	Simulink
Reshape	Simulink
Submatrix	DSP System Toolbox

Chirp

Purpose   Generate swept-frequency cosine (chirp) signal

Library   Sources

dspsrcs4
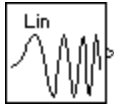
Description   The Chirp block outputs a swept-frequency cosine (chirp) signal with
unity amplitude and continuous phase. To specify the desired output
chirp signal, you must define its instantaneous frequency function, also
known as the output frequency sweep. The frequency sweep can be
linear, quadratic, or logarithmic, and repeats once every **Sweep time**
by default. See other sections of this reference page for more details
about the block.

Sections of This Reference Page

- Variables Used in This Reference Page on page 2-201
- "Setting the Output Frame Status" on page 2-201
- "Shaping the Frequency Sweep by Setting Frequency Sweep and
  Sweep Mode" on page 2-202
- "Unidirectional and Bidirectional Sweep Modes" on page 2-203
- "Setting Instantaneous Frequency Sweep Values" on page 2-204
- "Block Computation Methods" on page 2-205
- "Cautions Regarding the Swept Cosine Sweep" on page 2-208
- "Dialog Box" on page 2-210
- "Examples" on page 2-212
- "Supported Data Types" on page 2-223
- "See Also" on page 2-223

Variables Used in This Reference Page

f_0	**Initial frequency** parameter (Hz)
$f_i(t_g)$	**Target frequency** parameter (Hz)
t_g	**Target time** parameter (seconds)
T_{sw}	**Sweep time** parameter (seconds)
ϕ_0	**Initial phase** parameter (radians)
$\psi(t)$	Phase of the chirp signal (radians)
$f_i(t)$	User-specified output instantaneous frequency function (Hz); user-specified sweep
$f_{i(actual)}(t)$	Actual output instantaneous frequency function (Hz); actual output sweep
$y_{chirp}(t)$	Output chirp function

Setting the Output Frame Status

Use **Samples per frame** parameter to set the block's output frame status, as summarized in the following table. The **Sample time** parameter sets the sample time of both sample- and frame-based outputs.

Setting of Samples Per Frame Parameter	Output Frame Status
1	Sample based
n (any integer greater than 1)	Frame based, frame size n

Chirp

Shaping the Frequency Sweep by Setting Frequency Sweep and Sweep Mode

The basic shape of the output instantaneous frequency sweep, $f_i(t)$, is set by the **Frequency sweep** and **Sweep mode** parameters, described in the following table.

Parameters for Setting Sweep Shape	Possible Setting	Parameter Description
Frequency sweep	Linear Quadratic Logarithmic Swept cosine	Determines whether the sweep frequencies vary linearly, quadratically, or logarithmically. Linear and swept cosine sweeps both vary linearly.
Sweep mode	Unidirectional Bidirectional	Determines whether the sweep is unidirectional or bidirectional. For details, see "Unidirectional and Bidirectional Sweep Modes" on page 2-203

The following diagram illustrates the possible shapes of the frequency sweep that you can obtain by setting the **Frequency sweep** and **Sweep mode** parameters.

Possible Shapes of the Output Instantaneous Frequency Sweep

Swept Cosine and Linear Sweeps

Quadratic and Logarithmic Sweeps
Logarithmic sweeps cannot be negative.

These diagrams apply only to quadratic sweeps.

Positive Sweeps ($f_i(t_g) > f_0$)

Negative Sweeps ($f_i(t_g) < f_0$)

Positive Sweeps

Negative Sweeps

Unidirectional Sweeps

Bidirectional Sweeps
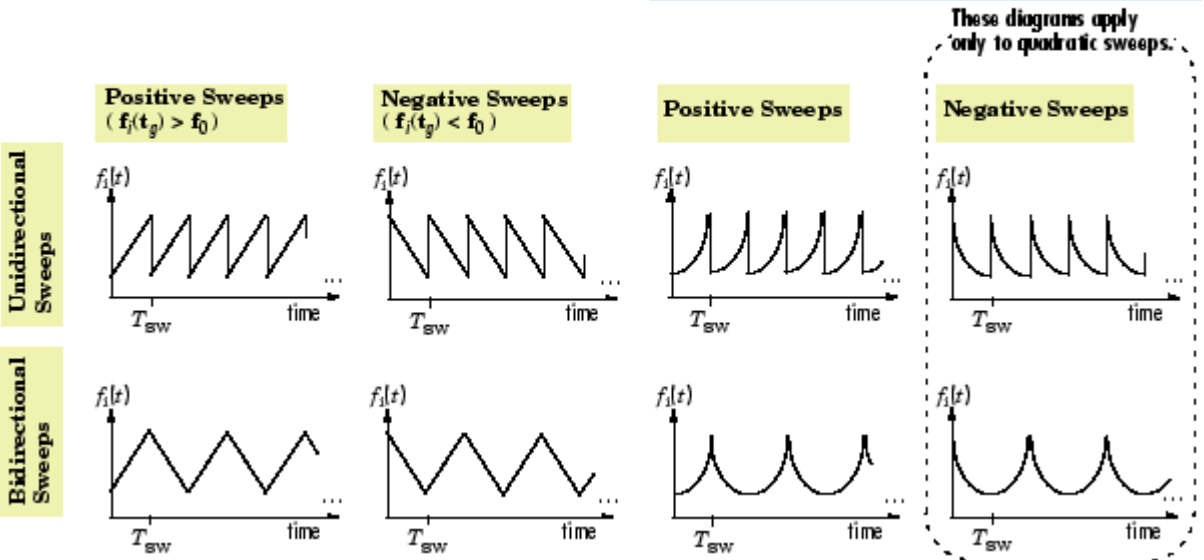
For information on how to set the frequency values in your sweep, see "Setting Instantaneous Frequency Sweep Values" on page 2-204.

Unidirectional and Bidirectional Sweep Modes

The **Sweep mode** parameter determines whether your sweep is unidirectional or bidirectional, which affects the shape of your output frequency sweep (see "Shaping the Frequency Sweep by Setting Frequency Sweep and Sweep Mode" on page 2-202). The following table describes the characteristics of unidirectional and bidirectional sweeps.

Chirp

Sweep Mode Parameter Settings	Sweep Characteristics
Unidirectional	• Lasts for one **Sweep time**, T_{sw} • Repeats once every T_{sw}
Bidirectional	• Lasts for twice the **Sweep time**, $2*T_{sw}$ • Repeats once every $2*T_{sw}$ • First half is identical to its unidirectional counterpart. • Second half is a mirror image of the first half.

The following diagram illustrates a linear sweep in both sweep modes. For information on setting the frequency values in your sweep, see "Setting Instantaneous Frequency Sweep Values" on page 2-204.

Unidirectional Linear Sweep   **Bidirectional Linear Sweep**
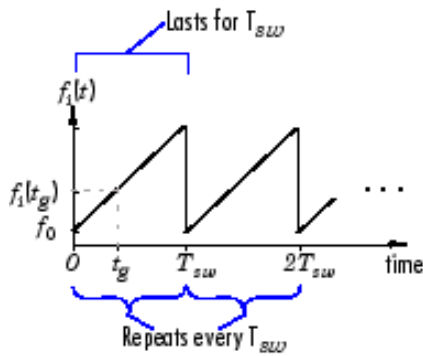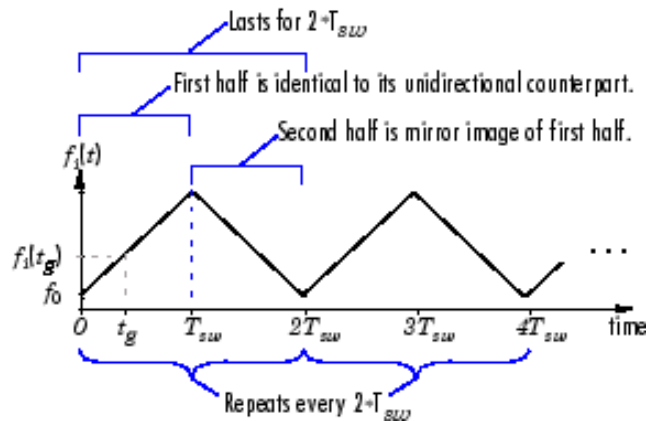
Setting Instantaneous Frequency Sweep Values

Set the following parameters to tune the frequency values of your output frequency sweep.

- **Initial frequency** (Hz), f_0

- **Target frequency** (Hz), $f_i(t_g)$

- **Target time** (seconds), t_g

The following table summarizes the sweep values at specific times for all **Frequency sweep** settings. For information on the formulas used to compute sweep values at other times, see "Block Computation Methods" on page 2-205.

Instantaneous Frequency Sweep Values

Frequency Sweep	Sweep Value at $t = 0$	Sweep Value at $t = t_g$	Time when Sweep Value Is Target Frequency, $f_i(t_g)$
Linear	f_0	$f_i(t_g)$	t_g
Quadratic	f_0	$f_i(t_g)$	t_g
Logarithmic	f_0	$f_i(t_g)$	t_g
Swept cosine	f_0	$2f_i(t_g) - f_0$	$t_g / 2$

Block Computation Methods

The Chirp block uses one of two formulas to compute the block output, depending on the **Frequency Sweep** parameter setting. For details, see the following sections:

- "Equations for Output Computation" on page 2-206

- "Output Computation Method for Linear, Quadratic, and Logarithmic Frequency Sweeps" on page 2-207

- "Output Computation Method for Swept Cosine Frequency Sweep" on page 2-208

Equations for Output Computation

The following table shows the equations used by the block to compute the user-specified output frequency sweep, $f_i(t)$, the block output, $y_{chirp}(t)$, and the actual output frequency sweep, $f_{i(actual)}(t)$. The only time the user-specified sweep is not the actual output sweep is when the **Frequency sweep** parameter is set to Swept cosine.

Note The following equations apply only to unidirectional sweeps in which $f_i(0) < f_i(t_g)$. To derive equations for other cases, you might find it helpful to examine the following table and the diagram in "Shaping the Frequency Sweep by Setting Frequency Sweep and Sweep Mode" on page 2-202.

The table below contains the following variables:

- $f_i(t)$ — the user-specified frequency sweep

- $f_{i(actual)}(t)$ — the actual output frequency sweep, usually equal to $f_i(t)$

- $y(t)$ — the Chirp block output

- $\psi(t)$ — the phase of the chirp signal, where $\psi(0) = 0$, and $2\pi f_i(t)$ is the derivative of the phase

$$f_i(t) = \frac{1}{2\pi} \cdot \frac{d\psi(t)}{dt}$$

- ϕ_0 — the **Initial phase** parameter value, where $y_{chirp}(0) = \cos(\phi_0)$

Equations Used by the Chirp Block for Unidirectional Positive Sweeps

Frequency Sweep	Block Output Chirp Signal	User-Specified Frequency Sweep, $f_i(t)$	β	Actual Frequency Sweep, $f_{i(actual)}(t)$
Linear	$y(t) = \cos(\psi(t) + \phi_0)$	$f_i(t) = f_0 + \beta t$	$\beta = \dfrac{f_i(t_g) - f_0}{t_g}$	$f_{i(actual)}^{(t)} = f_i^{(t)}$
Quadratic	Same as Linear	$f_i(t) = f_0 + \beta t^2$	$\beta = \dfrac{f_i(t_g) - f_0}{2}$	$f_{i(actual)}^{(t)} = f_i^{(t)}$
Logarithmic	Same as Linear	$F_i(t) = f_0 \left(\dfrac{f_i(t_g)}{f_0} \right)^{\frac{t}{t_g}}$ Where $f_i(t_g) > f_0 > 0$	N/A	$f_{i(actual)}(t) = f_i(t)$
Swept cosine	$y(t) = \cos(2\pi f_i(t)t + \phi_0)$	Same as Linear	Same as Linear	$f_{i(actual)}(t) = f_i(t) + \beta t$

Output Computation Method for Linear, Quadratic, and Logarithmic Frequency Sweeps

The derivative of the phase of a chirp function gives the instantaneous frequency of the chirp function. The Chirp block uses this principle to calculate the chirp output when the **Frequency Sweep** parameter is set to Linear, Quadratic, or Logarithmic.

$$y_{chirp}(t) = \cos(\psi(t) + \phi_0)$$

Linear, quadratic, or logarithmic chirp signal with phase $\psi(t)$

$$f_i(t) = \frac{1}{2\pi} \cdot \frac{d\psi(t)}{dt}$$

Phase derivative is instantaneous frequency

For instance, if you want a chirp signal with a linear instantaneous frequency sweep, you should set the **Frequency Sweep** parameter to

Linear, and tune the linear sweep values by setting other parameters appropriately. The block outputs a chirp signal, the phase derivative of which is the specified linear sweep. This ensures that the instantaneous frequency of the output is the linear sweep you desired. For equations describing the linear, quadratic, and logarithmic sweeps, see "Equations for Output Computation" on page 2-206.

Output Computation Method for Swept Cosine Frequency Sweep

To generate the swept cosine chirp signal, the block sets the swept cosine chirp output as follows.

$$y_{chirp}(t) = \cos(\psi(t) + \phi_0) = \cos(2\pi f_i(t)t + \phi_0)$$

Swept cosine chirp output (Instantaneous frequency equation, shown above, does not hold.)

Note that the instantaneous frequency equation, shown above, does not hold for the swept cosine chirp, so the user-defined frequency sweep, $f_i(t)$, is not the actual output frequency sweep, $f_{i(actual)}(t)$, of the swept cosine chirp. Thus, the swept cosine output might not behave as you expect. To learn more about swept cosine chirp behavior, see "Cautions Regarding the Swept Cosine Sweep" on page 2-208 and "Equations for Output Computation" on page 2-206.

Cautions Regarding the Swept Cosine Sweep

When you want a linearly swept chirp signal, we recommend you use a linear frequency sweep. Though a swept cosine frequency sweep also yields a linearly swept chirp signal, the output might have unexpected frequency content. For details, see the following two sections.

Swept Cosine Instantaneous Output Frequency at the Target Time is not the Target Frequency

The swept cosine sweep value at the **Target time** is not necessarily the **Target frequency**. This is because the user-specified sweep is not the

actual frequency sweep of the swept cosine output, as noted in "Output Computation Method for Swept Cosine Frequency Sweep" on page 2-208. See the table Instantaneous Frequency Sweep Values on page 2-205 for the actual value of the swept cosine sweep at the **Target time**.

Swept Cosine Output Frequency Content May Greatly Exceed Frequencies in the Sweep

In **Swept cosine** mode, you should not set the parameters so that $1/T$sw is very large compared to the values of the **Initial frequency** and **Target frequency** parameters. In such cases, the actual frequency content of the swept cosine sweep might be closer to $1/T$sw, far exceeding the **Initial frequency** and **Target frequency** parameter values.

Chirp
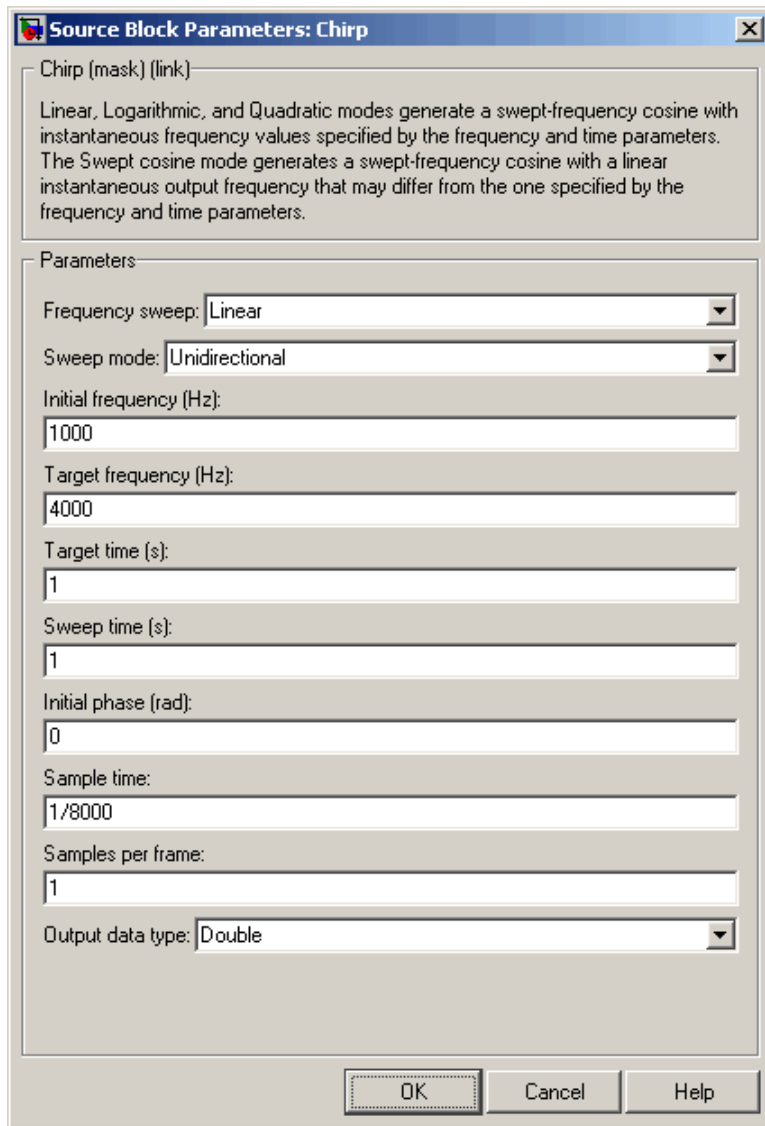
**Dialog
Box**

Frequency sweep

The type of output instantaneous frequency sweep, $f_i(t)$: Linear, Logarithmic, Quadratic, or Swept cosine.

Sweep mode

> The directionality of the chirp signal: `Unidirectional` or `Bidirectional`.

Initial frequency (Hz)

> For `Linear`, `Quadratic`, and `Swept cosine` sweeps, the initial frequency, f_0, of the output chirp signal. For `Logarithmic` sweeps, **Initial frequency** is one less than the actual initial frequency of the sweep. Also, when the sweep is `Logarithmic`, you must set the **Initial frequency** to be less than the **Target frequency**. Tunable.

Target frequency (Hz)

> For `Linear`, `Quadratic`, and `Logarithmic` sweeps, the instantaneous frequency, $f_i(t_g)$, of the output at the **Target time**, t_g. For a `Swept cosine` sweep, **Target frequency** is the instantaneous frequency of the output at half the **Target time**, $t_g/2$. When **Frequency sweep** is `Logarithmic`, you must set the **Target frequency** to be greater than the **Initial frequency**. Tunable.

Target time (s)

> For `Linear`, `Quadratic`, and `Logarithmic` sweeps, the time, t_g, at which the **Target frequency**, $f_i(t_g)$, is reached by the sweep. For a `Swept cosine` sweep, **Target time** is the time at which the sweep reaches $2f_i(t_g) - f_0$. You must set **Target time** to be *no*
>
> *greater than* **Sweep time** , $T_{sw} \geq t_g$. Tunable.

Sweep time (s)

> In `Unidirectional` **Sweep mode**, the **Sweep time**, T_{sw}, is the period of the output frequency sweep. In `Bidirectional` **Sweep mode**, the **Sweep time** is half the period of the output frequency sweep. You must set **Sweep time** to be no less than **Target**
>
> **time**, $T_{sw} \geq t_g$. Tunable.

Initial phase (rad)

> The phase, ϕ_0 , of the cosine output at $t=0$; $y_{chirp}(t) = \cos(\phi_0)$. Tunable.

Chirp

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. When the value of this parameter is 1, the block outputs a sample-based signal.

Output data type

> The data type of the output, single-precision or double-precision.

Examples   The first few examples demonstrate how to use the Chirp block's main parameters, how to view the output in the time domain, and how to view the output spectrogram:

- "Example 1: Setting a Final Frequency Value for Unidirectional Sweeps" on page 2-212

- "Example 2: Bidirectional Sweeps" on page 2-216

- "Example 3: When Sweep Time is Greater Than Target Time" on page 2-218

Examples 4 and 5 illustrate Chirp block settings that might produce unexpected outputs:

- "Example 4: Output Sweep with Negative Frequencies" on page 2-220

- "Example 5: Output Sweep with Frequencies Greater Than Half the Sampling Frequency" on page 2-221

Example 1: Setting a Final Frequency Value for Unidirectional Sweeps

Often times, you might want a unidirectional sweep for which you know the initial and final frequency values. You can specify the final frequency of a unidirectional sweep by setting **Target time** equal to **Sweep time**, in which case the **Target frequency** becomes the final frequency in the sweep. The following model demonstrates this method.

This technique might not work for swept cosine sweeps. For details, see "Cautions Regarding the Swept Cosine Sweep" on page 2-208.

Open the Example 1 model by typing ex_chirp_ref at the MATLAB command line. You can also rebuild the model yourself; see the following list for model parameter settings (leave unlisted parameters in their default states).
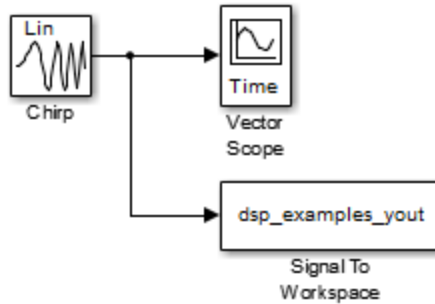
Since **Target time** is set to equal **Sweep time** (1 second), the **Target frequency** (25 Hz) is the final frequency of the unidirectional sweep. Run your model to see the time domain output:
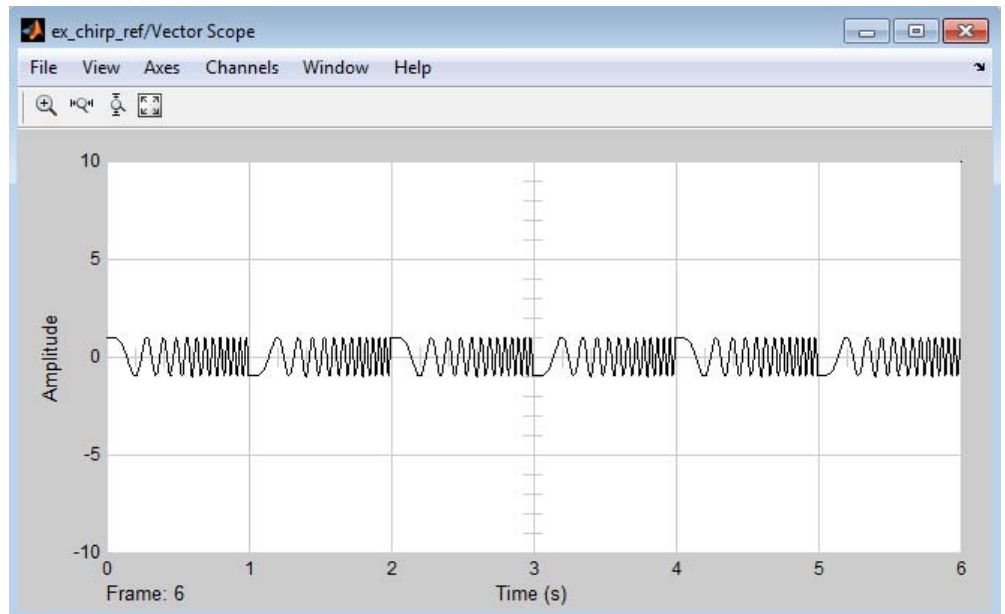
Chirp

Type the following command to view the chirp output spectrogram:

```
spectrogram(dsp_examples_yout,hamming(128),...
    110,[0:.01:40],400)
```
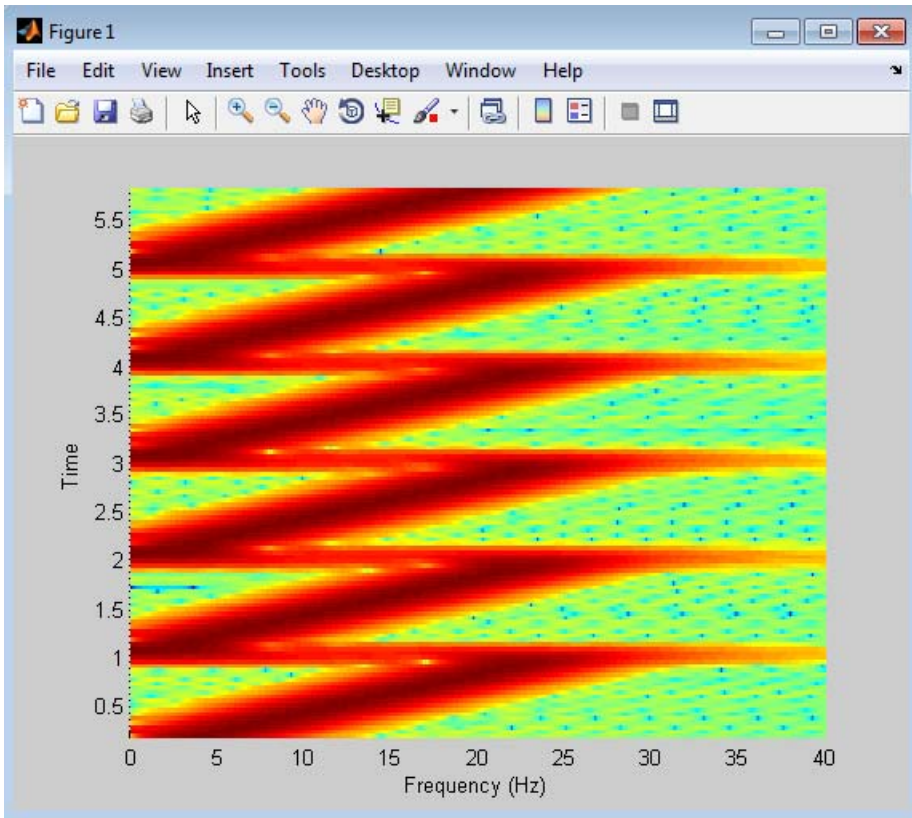
Chirp Block Parameters for Example 1	
Frequency sweep	Linear
Sweep mode	Unidirectional
Initial frequency	0
Target frequency	25
Target time	1
Sweep time	1

Initial phase	0
Sample time	1/400
Samples per frame	400
Vector Scope Block Parameters for Example 1	
Input domain	`Time`
Time display span	6
Signal To Workspace Block Parameters for Example 1	
Variable name	`dsp_examples_yout`
Configuration Dialog Parameters for Example 1	
Stop time	5

Example 2: Bidirectional Sweeps

Change the **Sweep mode** parameter in the Example 1 model to
Bidirectional, and leave all other parameters the same to view the
following bidirectional chirp. Note that in the bidirectional sweep, the
period of the sweep is twice the **Sweep time** (2 seconds), whereas it was
one **Sweep time** (1 second) for the unidirectional sweep in Example 1.

Open the Example 2 model by typing ex_chirp_ref2 at the MATLAB
command line.
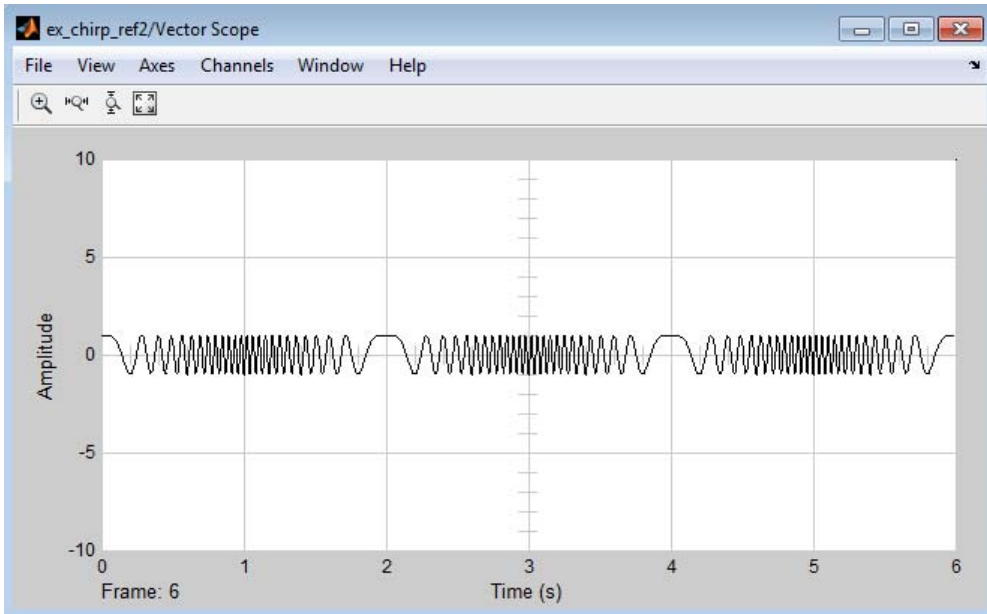
Run your model to see the time domain output:

Type the following command to view the chirp output spectrogram:

```
spectrogram(dsp_examples_yout,hamming(128),...
    110,[0:.01:40],400)
```
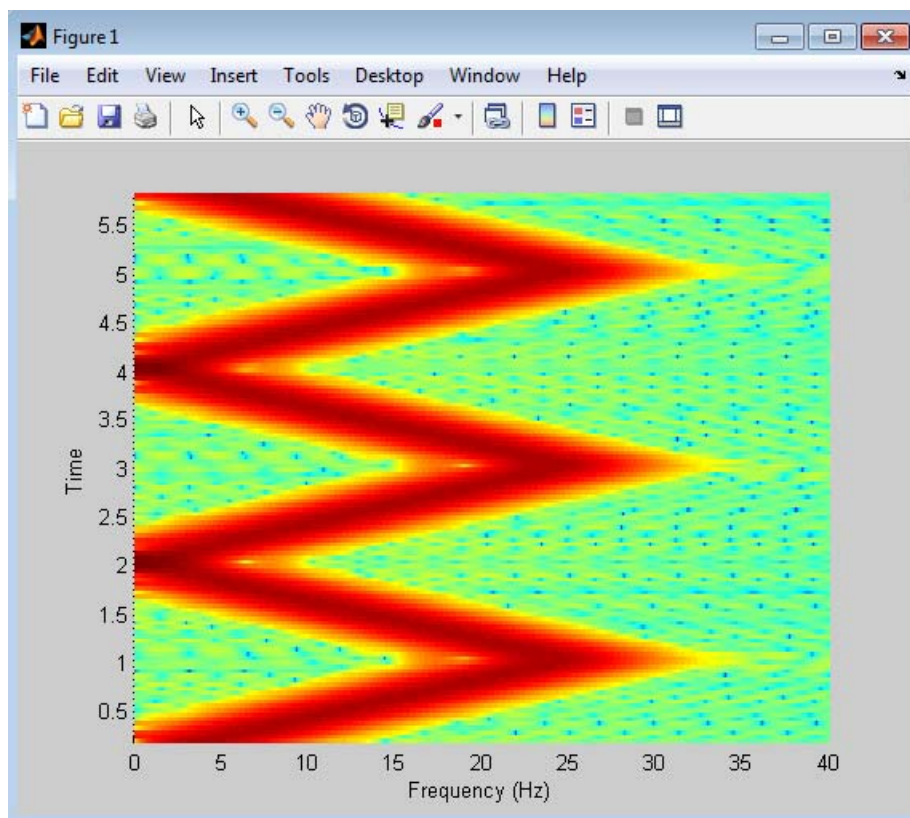
Example 3: When Sweep Time is Greater Than Target Time

Setting **Sweep time** to 1.5 and leaving the rest of the parameters as in the Example 1 model gives the following output. The sweep still reaches the **Target frequency** (25 Hz) at the **Target time** (1 second), but since **Sweep time** is greater than **Target time**, the sweep continues on its linear path until one **Sweep time** (1.5 seconds) is traversed.

Unexpected behavior might arise when you set **Sweep time** greater than **Target time**; see "Example 4: Output Sweep with Negative Frequencies" on page 2-220 for details.

Open the Example 3 model by typing `ex_chirp_ref3` at the MATLAB command line.
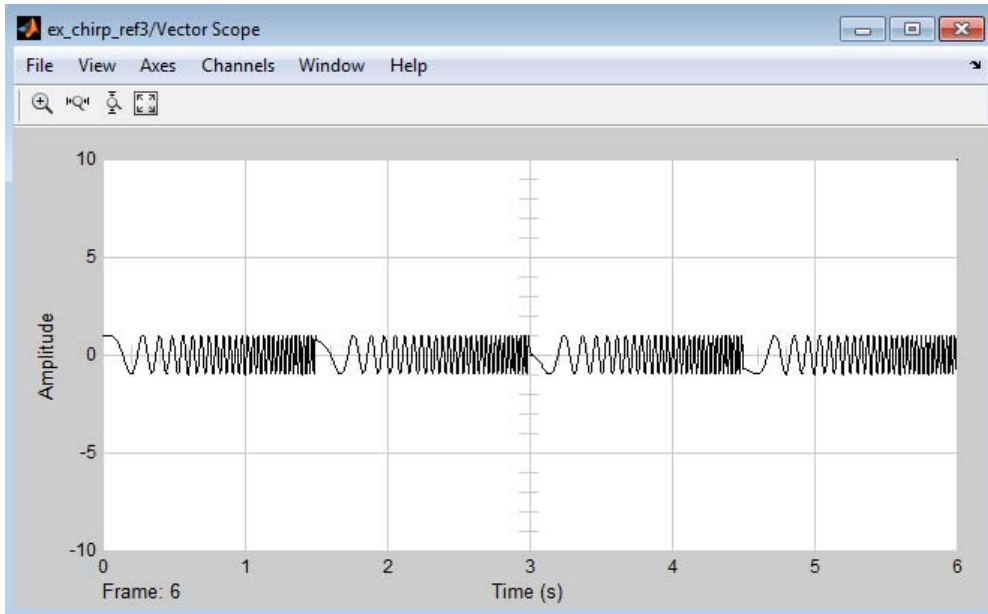
Run your model to see the time domain output:

Type the following command to view the chirp output spectrogram:

```
spectrogram(dsp_examples_yout,hamming(128),...
     110,[0:.01:40],400)
```
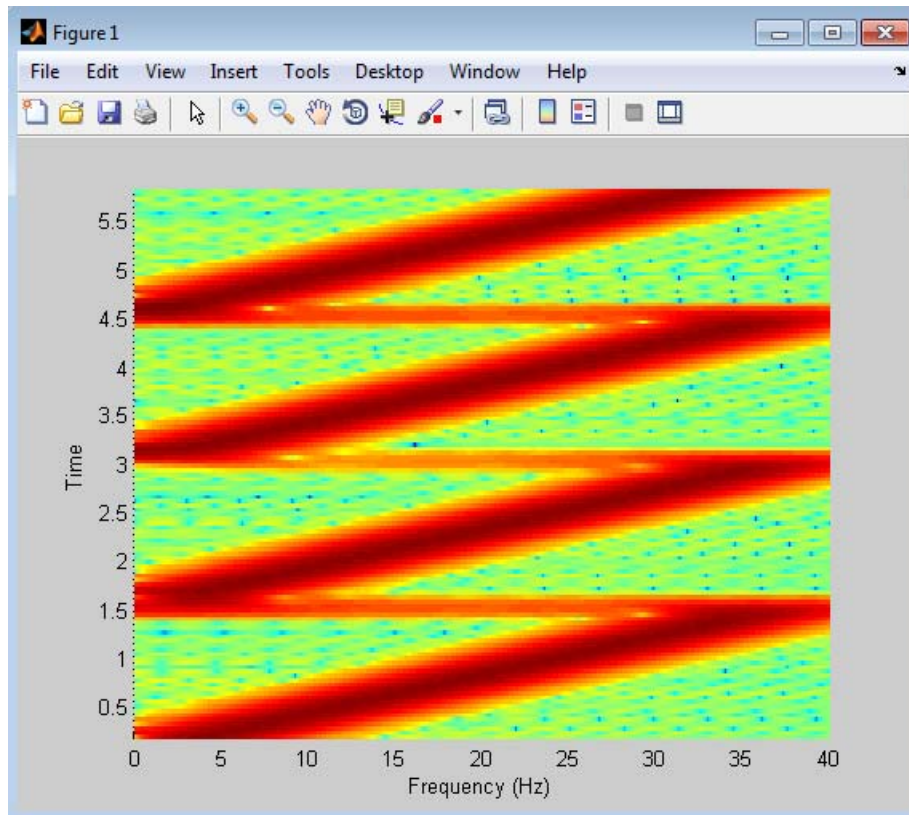
Example 4: Output Sweep with Negative Frequencies

Modify the Example 1 model by changing **Sweep time** to 1.5, **Initial frequency** to 25, and **Target frequency** to 0. *The output chirp of this example might not behave as you expect* because the sweep contains negative frequencies between 1 and 1.5 seconds. The sweep reaches the **Target frequency** of 0 Hz at one second, then continues on its negative slope, taking on negative frequency values until it traverses one **Sweep time** (1.5 seconds).

Open the Example 4 model by typing ex_chirp_ref4 at the MATLAB command line.

Run your model to see the time domain output:
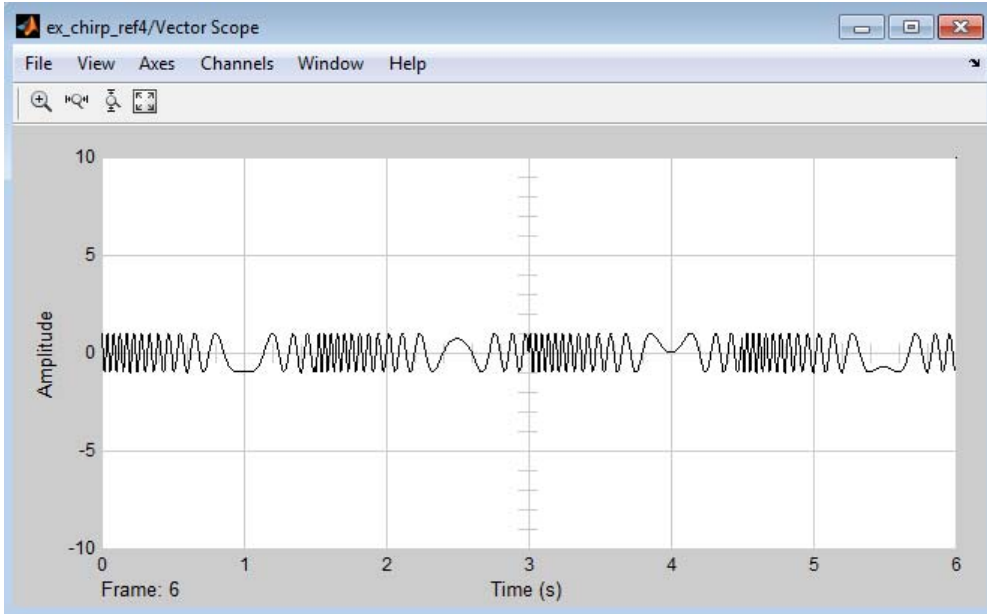
Example 5: Output Sweep with Frequencies Greater Than Half the Sampling Frequency

Modify the Example 1 model by changing the **Target frequency** parameter to 275. *The output chirp of this model might not behave as you expect* because the sweep contains frequencies greater than half the sampling frequency (200 Hz), which causes aliasing. If you unexpectedly get a chirp output with a spectrogram resembling the one following, your chirp's sweep might contain frequencies greater than half the sampling frequency.

Open the Example 5 model by typing ex_chirp_ref5 at the MATLAB command line.
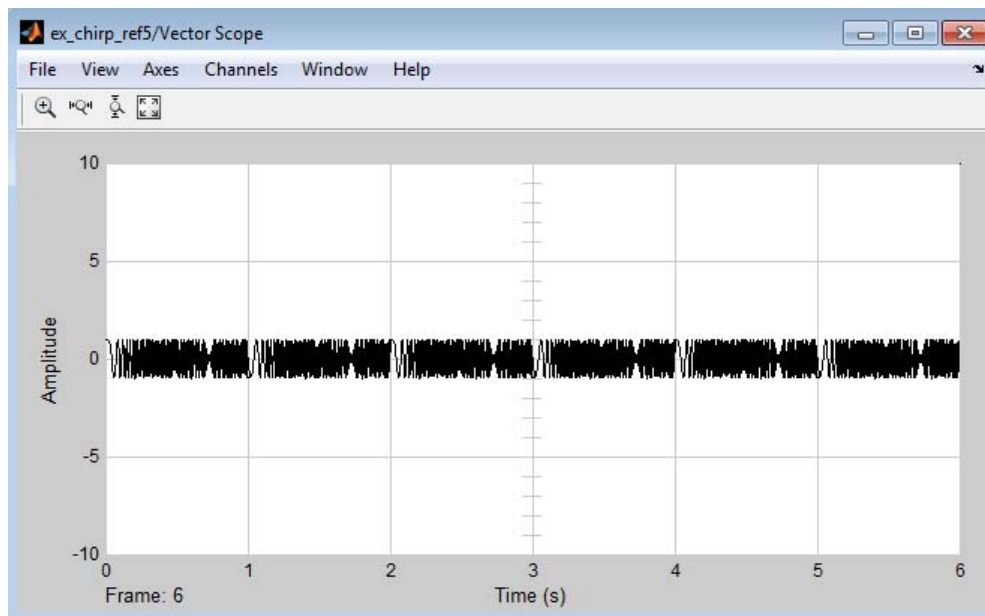
Run your model to see the time domain output:

Chirp

Type the following command to view the chirp output spectrogram:

```
spectrogram(dsp_examples_yout,hamming(64),...
    60,256,400)
```
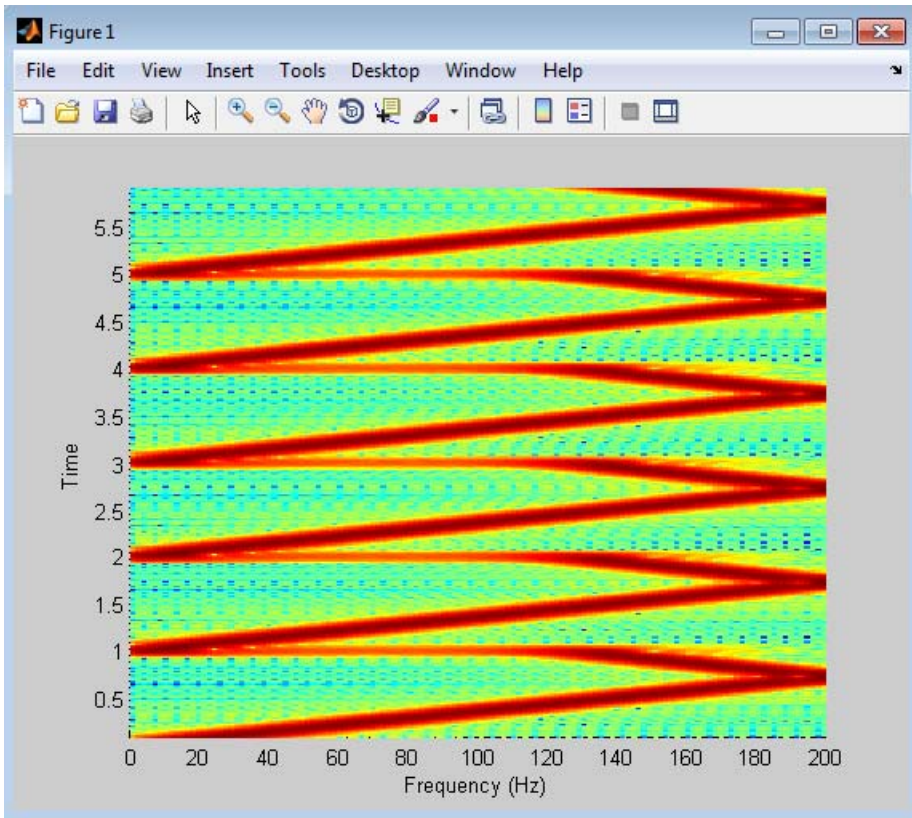
| **Supported Data Types** | • Double-precision floating point |
| | • Single-precision floating point |

| **See Also** | Signal From Workspace | DSP System Toolbox |
| | Signal Generator | Simulink |

Chirp

Sine Wave	DSP System Toolbox
chirp	Signal Processing Toolbox
spectrogram	Signal Processing Toolbox

Purpose   Factor square Hermitian positive definite matrix into triangular components

Library   Math Functions / Matrices and Linear Algebra / Matrix Factorizations

dspfactors

Description

The Cholesky Factorization block uniquely factors the square Hermitian positive definite input matrix S as

$$S = LL^*$$
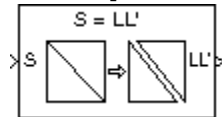
where *L* is a lower triangular square matrix with positive diagonal elements and L^* is the Hermitian (complex conjugate) transpose of *L*. The block outputs a matrix with lower triangle elements from *L* and upper triangle elements from L^*. The output is not in the same form as the output of the MATLAB chol function. In order to convert the output of the Cholesky Factorization block to the MATLAB form, use the following equation:

```
R = triu(LL');
```

Here, LL' is the output of the Cholesky Factorization block. Due to roundoff error, these equations do not produce a result that is exactly the same as the MATLAB result.

Cholesky Factorization

$$\begin{bmatrix} 9 & -1 & 2 \\ -1 & 8 & -5 \\ 2 & -5 & 7 \end{bmatrix}$$
$$S$$

$$S = LL'$$
$$S \boxed{} \Rightarrow \boxed{} LL'$$
Cholesky
Factorization

$$\begin{bmatrix} 3.00 & -0.33 & 0.67 \\ -0.33 & 2.81 & -1.70 \\ 0.67 & -1.70 & 1.91 \end{bmatrix}$$
$$LL'$$

$$L = \begin{bmatrix} 3.00 & 0 & 0 \\ -0.33 & 2.81 & 0 \\ 0.67 & -1.70 & 1.91 \end{bmatrix}$$

Output matrix *LL'* with lower triangular elements from *L* and upper triangular elements from *L**.
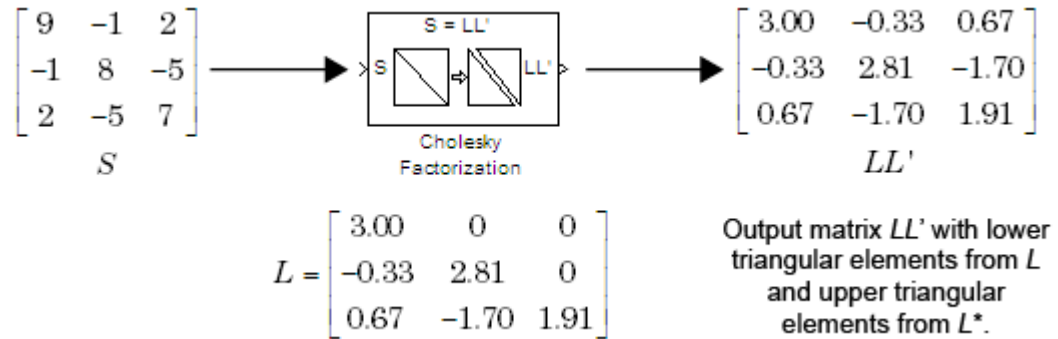
Block Output Composed of L and L*

Input Requirements for Valid Output

The block output is valid only when its input has the following characteristics:

- Hermitian — The block does *not* check whether the input is Hermitian; it uses only the diagonal and upper triangle of the input to compute the output.

- Real-valued diagonal entries — The block disregards any imaginary component of the input's diagonal entries.

- Positive definite — Set the block to notify you when the input is not positive definite as described in "Response to Nonpositive Definite Input" on page 2-226.

Response to Nonpositive Definite Input

To generate a valid output, the block algorithm requires a positive definite input (see "Input Requirements for Valid Output" on page 2-226). Set the **Non-positive definite input** parameter to determine how the block responds to a nonpositive definite input:

- Ignore — Proceed with the computation and *do not* issue an alert. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.

- `Warning` — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.

- `Error` — Display an error dialog and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to `Ignore` in the code generated for this block by Simulink Coder code generation software.

Performance Comparisons with Other Blocks

Note that L and L^* share the same diagonal in the output matrix. Cholesky factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable.
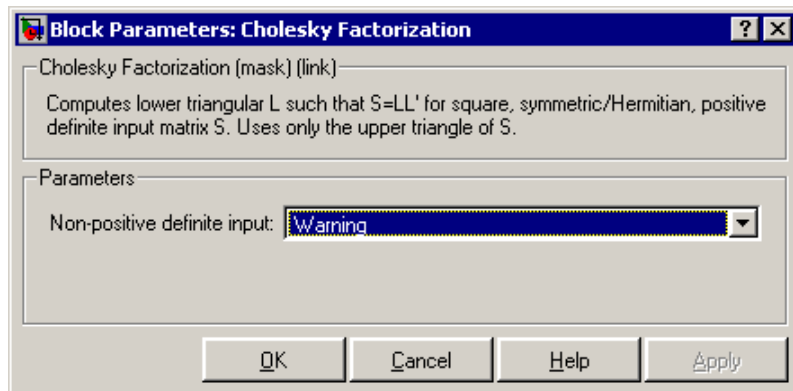
Dialog Box

Non-positive definite input

Response to nonpositive definite matrix inputs: `Ignore`, `Warning`, or `Error`. See "Response to Nonpositive Definite Input" on page 2-226.

Cholesky Factorization

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed.
Baltimore, MD: Johns Hopkins University Press, 1996.

**Supported
Data
Types**

Port	Supported Data Types
S	• Double-precision floating point • Single-precision floating point
LL'	• Double-precision floating point • Single-precision floating point

See Also

Autocorrelation LPC	DSP System Toolbox
Cholesky Inverse	DSP System Toolbox
Cholesky Solver	DSP System Toolbox
LDL Factorization	DSP System Toolbox
LU Factorization	DSP System Toolbox
QR Factorization	DSP System Toolbox
chol	MATLAB

See "Matrix Factorizations" for related information.

Purpose   Compute inverse of Hermitian positive definite matrix using Cholesky factorization

Library   Math Functions / Matrices and Linear Algebra / Matrix Inverses

dspinverses
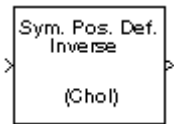
Description   The Cholesky Inverse block computes the inverse of the Hermitian positive definite input matrix S by performing Cholesky factorization.

$$S^{-1} = (LL^*)^{-1}$$

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.

Response to Nonpositive Definite Input

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.

Cholesky Inverse

> **Note** The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to `Ignore` in the code generated for this block by Simulink Coder code generation software.
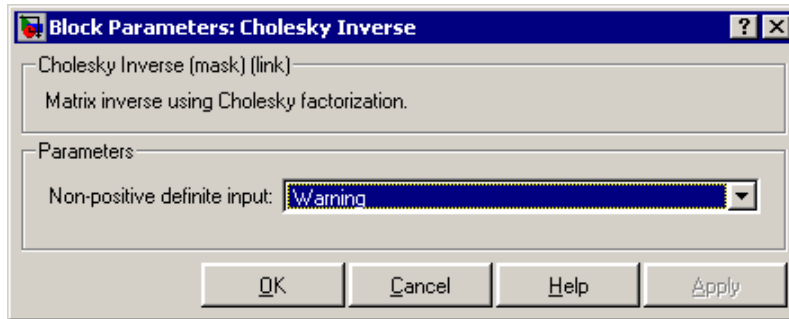
Dialog Box

Non-positive definite input
Response to nonpositive definite matrix inputs: `Ignore`, `Warning`, or `Error`. See "Response to Nonpositive Definite Input" on page 2-229.

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

See Also

Cholesky Factorization	DSP System Toolbox
Cholesky Solver	DSP System Toolbox
LDL Inverse	DSP System Toolbox

LU Inverse	DSP System Toolbox
Pseudoinverse	DSP System Toolbox
inv	MATLAB

See "Matrix Inverses" for related information.

Cholesky Solver

Purpose   Solve *SX=B* for X when *S* is square Hermitian positive definite matrix

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers

dspsolvers
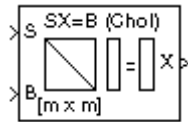
Description   The Cholesky Solver block solves the linear system *SX=B* by applying
Cholesky factorization to input matrix at the S port, which must be
square (*M*-by-*M*) and Hermitian positive definite. Only the diagonal and
upper triangle of the matrix are used, and any imaginary component
of the diagonal entries is disregarded. The input to the B port is the
right side *M*-by-*N* matrix, *B*. The *M*-by-*N* output matrix X is the unique
solution of the equations.

A length-*M* vector input for right side *B* is treated as an *M*-by-1 matrix.

Response to Nonpositive Definite Input

When the input is not positive definite, the block reacts with the
behavior specified by the **Non-positive definite input** parameter.
The following options are available:

- Ignore — Proceed with the computation and *do not* issue an alert.
  The output is *not* a valid solution.

- Warning — Proceed with the computation and display a warning
  message in the MATLAB Command Window. The output is *not* a
  valid solution.

- Error — Display an error dialog box and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic
parameter. Like all diagnostic parameters on the Configuration
Parameters dialog box, it is set to Ignore in the code generated for this
block by Simulink Coder code generation software.

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 $SX=B$ 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
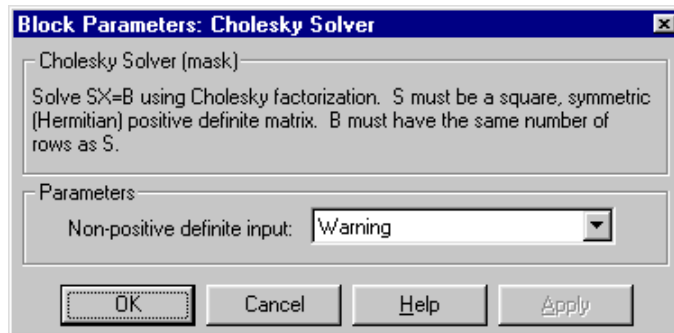
Non-positive definite input
> Response to nonpositive definite matrix inputs: Ignore, Warning, or Error. See "Response to Nonpositive Definite Input" on page 2-232.

Cholesky Solver

Supported Data Types

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

See Also

Autocorrelation LPC	DSP System Toolbox
Cholesky Factorization	DSP System Toolbox
Cholesky Inverse	DSP System Toolbox
LDL Solver	DSP System Toolbox
LU Solver	DSP System Toolbox
QR Solver	DSP System Toolbox
chol	MATLAB

See "Linear System Solvers" for related information.

Purpose   Design CIC compensator

Library   Filtering / Filter Designs

                       dspfdesign

Description   This block brings the filter design capabilities of the `filterbuilder`
                       function to the Simulink environment.
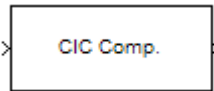
> CIC Comp. >

**Dialog
Box**   See "CIC Compensator Filter Design Dialog Box — Main Pane" on page
                       7-696 for more information about the parameters of this block. The
                       Data Types and **Code Generation** panes are not available for blocks
                       in the DSP System Toolbox Filter Designs library.

CIC Compensator

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the filter order mode and the filter type.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

CIC Compensator

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Number of CIC sections

Specify the number of sections in the CIC filter for which you are designing this compensator. Select the number of sections from the drop-down list or enter the number.

Differential Delay

Specify the differential delay of your target CIC filter. The default value is 1. Most CIC filters use 1 or 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Frequency Specifications

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you

provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the

specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default method is `Equiripple`.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select passband or stopband or both from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- Flat — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- Linear — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- 1/f — Specifies that the stopband attenuation changes exponentially as the frequency increases, where f is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to Linear, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.

- When you set **Stopband shape** to 1/f, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. The block applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

CIC Compensator

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, the filter uses direct-form structure.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

> **Note** The `Inherited (this choice will be removed   see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.

- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

CIC Decimation

Purpose　　　　Decimate signal using Cascaded Integrator-Comb filter

Library　　　　Filtering / Multirate Filters

dspmlti4
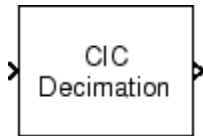
Description　　　The CIC Decimation block performs a sample rate decrease (decimation) on an input signal by an integer factor. Cascaded Integrator-Comb (CIC) filters are a class of linear phase FIR filters comprised of a comb part and an integrator part.

The transfer function of a CIC decimator filter is

$$H(z) = H_I{}^N(z)H_c{}^N(z) = \frac{\left(1 - z^{-RM}\right)^N}{\left(1 - z^{-1}\right)^N} = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N$$

where

- H_I is the transfer function of the integrator part of the filter.

- H_C is the transfer function of the comb part of the filter.

- N is the number of sections. The number of sections in a CIC filter is defined as the number of sections in either the comb part *or* the integrator part of the filter. This value does not represent the total number of sections throughout the entire filter.

- R is the decimation factor.

- M is the differential delay.

The CIC Decimation block requires a Simulink Fixed Point license. The block supports real and complex fixed-point inputs and treats each channel of a complex input as two real input channels.

CIC Filter Structure

The filter structures supported by the CIC Decimation and CIC Interpolation blocks exactly match those created by the mfilt.cicdecim

and `mfilt.cicinterp` objects. You can create an `mfilt` object in any workspace and specify that object in the **Multirate filter variable** parameter of this block. Alternatively, you can set the **Coefficient source** to **Dialog parameters** and specify information about the CIC filter using the available block dialog parameters.

You can use this block to create the following CIC filter structure. This decimator has a latency of N, where N is the number of sections in either the comb or the integrator part of the filter.
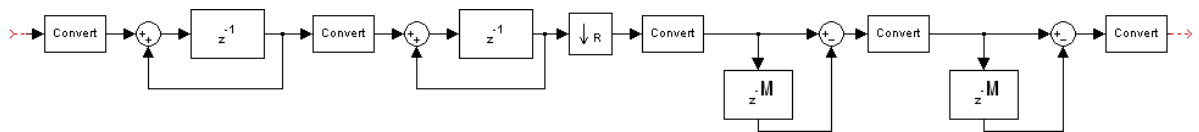
Examples   See GSM Digital Down Converter for an example using the CIC Decimation block.

Dialog Box

Coefficient Source

The CIC Decimation block can operate in two different modes. Select the mode in the **Coefficient source** group box. If you select

- **Dialog parameters**, you enter information about the filter such as structure and coefficients in the block mask.

- **Multirate filter object (MFILT)**, you specify the filter using an `mfilt` object.

Different items appear on the CIC Decimation block dialog depending on whether you select **Dialog parameters** or **Multirate filter object (MFILT)** in the **Coefficient source** group box. See the following sections for details:

- "Specify Filter Characteristics in Dialog" on page 2-246

- "Specify Multirate Filter Object" on page 2-250

CIC Decimation

Specify Filter Characteristics in Dialog

The **Main** pane of the CIC Decimation block dialog appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.
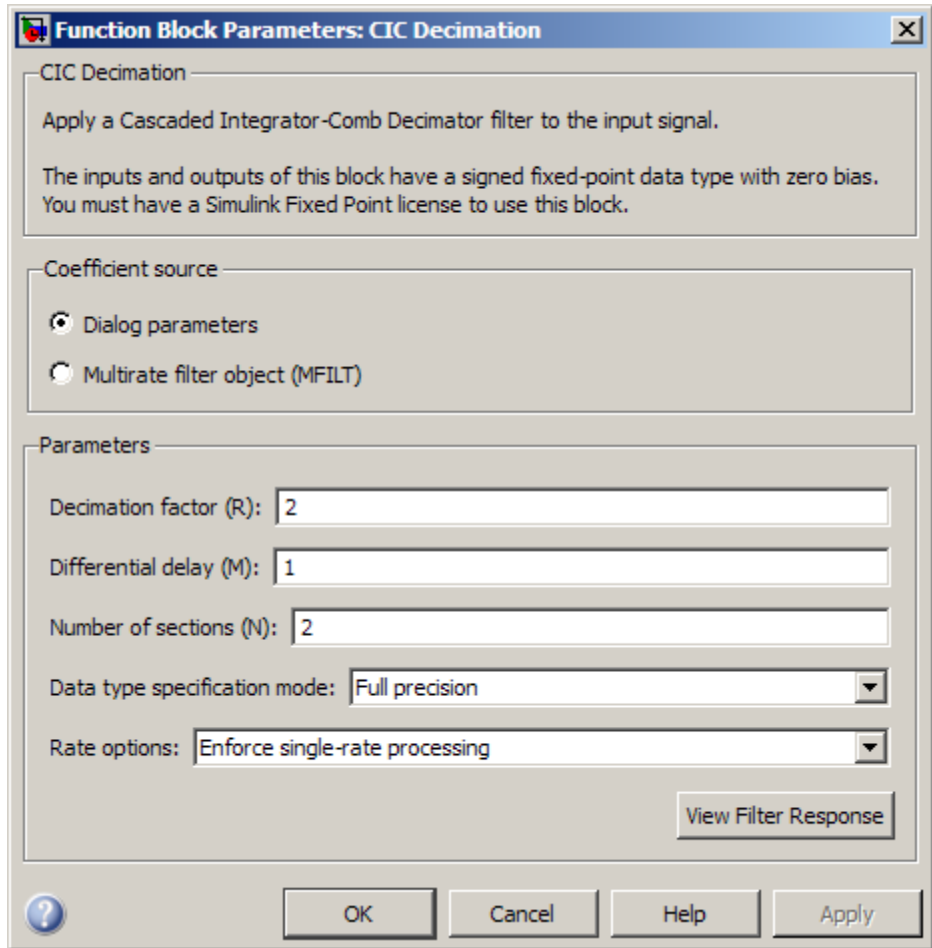
Decimation factor (R)

Specify the decimation factor of the filter.

Differential delay (M)

Specify the differential delay of the comb part of the filter, M, as shown in the diagram in "CIC Filter Structure" on page 2-244.

Number of sections (N)

Specify the number of filter sections. The number you specify determines the number of sections in either the comb part of the filter or the integrator part of the filter. This value does not represent the total number of sections in the comb and integrator parts combined.

Data type specification mode

Choose how you specify the fixed-point word length and fraction length of the filter sections and/or output.

- `Full precision` — In this mode, the word and fraction lengths of the filter sections and outputs are automatically selected for you. All word lengths are set to

$$\text{word length} = \text{ceil}(N * \log_2(M * R)) + I$$

where

  — I = input word length

  — M = differential delay

  — N = number of sections

  — R = decimation factor

All fraction lengths are set to the input fraction length.

- `Minimum section word lengths` — In this mode, you specify the word length of the filter output in the **Output word length** parameter. The block automatically selects the word lengths of the filter sections and all fraction lengths such that each of the section word lengths is as small as possible. The

precision of each filter section is less than in `Full precision` mode, but the range of each section is preserved.

- `Specify word lengths` — In this mode you specify the word lengths of the filter sections and output in the **Section word lengths** and **Output word length** parameters. The block automatically selects fraction lengths for the filter sections and output such that the range of each section is preserved when the least significant bits are discarded.

- `Binary point scaling` — In this mode you fully specify the word and fraction lengths of the filter sections and output in the **Section word lengths**, **Section fraction lengths**, **Output word length**, and **Output fraction length** parameters.

Section word lengths

Specify the word length, in bits, of the filter sections.

This parameter is only visible if `Specify word lengths` or `Binary point scaling` is selected for the **Data type specification mode** parameter.

Section fraction lengths

Specify the fraction length of the filter sections.

This parameter is only visible if `Binary point scaling` is selected for the **Data type specification mode** parameter.

Output word length

Specify the word length, in bits, of the filter output.

This parameter is only visible if you select `Minimum section word lengths`, `Specify word lengths`, or `Binary point scaling` for the **Data type specification mode** parameter.

Output fraction length

Specify the fraction length of the filter output.

This parameter is only visible if `Binary point scaling` is selected for the **Data type specification mode** parameter.

Rate options

Specify the rate processing rule for the block. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block performs frame-based processing and produces an output that has the same sample rate as the input. To decimate the signal while maintaining the input sample rate, the block decreases the output frame size. In this mode, the input column size must be a multiple of the decimation factor, `R`.

- `Allow multirate processing` — When you select this option, the block performs sample-based processing. In this mode, the block produces an output with a sample rate that is `R` times slower than the input sample rate.

Note The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note If you specify an `mfilt` object in the **Multirate filter variable** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

CIC Decimation

Specify Multirate Filter Object

The **Main** pane of the CIC Decimation block dialog appears as follows when you select **Multirate filter object (MFILT)** in the **Coefficient source** group box.

Multirate filter variable

Specify the multirate filter object (`mfilt`) that you would like the block to implement. You can specify the `mfilt` object in one of three ways:

- You can fully specify the `mfilt` object in the block mask.

- You can enter the variable name of a `mfilt` object that is defined in any workspace.

- You can enter a variable name for a `mfilt` object that is not yet defined, as shown in the default value.

For more information on creating `mfilt` objects, see the `mfilt` function reference page.

Rate options

Specify the rate processing rule for the block. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block performs frame-based processing and produces an output that has the same sample rate as the input. To decimate the signal while maintaining the input sample rate, the block decreases the output frame size. In this mode, the input column size must be a multiple of the decimation factor, `R`.

- `Allow multirate processing` — When you select this option, the block performs sample-based processing and produces an output with a sample rate that is `R` times slower than the input sample rate.

Note The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

CIC Decimation

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the `mfilt` object specified in the **Multirate filter variable** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note If you specify an `mfilt` object in the **Multirate filter variable** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

References

[1] Hogenauer, E.B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-29(2): pp. 155–162, 1981.

[2] Meyer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, Springer Verlag, 2001.

[3] Harris, Fredric J., *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

Supported Data Types

- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

CIC Interpolation	DSP System Toolbox
FIR Decimation	DSP System Toolbox
FIR Interpolation	DSP System Toolbox
filter	DSP System Toolbox

CIC Filter

Purpose	Design Cascaded Integrator-Comb (CIC) Filter
Library	Filtering / Filter Designs
	dspfdesign

Description

This block brings the filter design capabilities of the filterbuilder function to the Simulink environment.

Dialog Box

Main Pane

See "CIC Filter Design Dialog Box — Main Pane" on page 7-692 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Function Block Parameters: CIC Filter

CIC Filter

Design a Cascaded Integrator-Comb filter.

Main	Data Types

Filter specifications

Filter Type: Decimator   Factor: 2

Differential delay: 1

Frequency specifications

Frequency units: Normalized (0 to 1)   Input Fs: 2

Fpass: .01

Magnitude specifications

Magnitude units: dB

Astop: 60

Filter Implementation

☐ Use basic elements to enable filter customization

Rate options: Enforce single-rate processing

OK   Cancel   Hel

CIC Filter

View filter response

> This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

> - Magnitude response, phase response, and group delay in the frequency domain.
>
> - Impulse response and step response in the time domain.
>
> - Pole-zero information.
>
> The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your CIC filter format, such as the filter type and the differential delay.

Filter type

> Select whether your filter will be a `decimator` or an `interpolator`. Your choice determines the type of filter and the design methods and structures that are available to implement your filter. Selecting `decimator` or `interpolator` activates the **Factor** option. When you design an interpolator, you enable the **Output Fs** parameter.

> When you design either a decimator or interpolator, the resulting filter is a CIC filter that decimates or interpolates your input signal.

Differential delay

> Specify the differential delay of your CIC filter as an integer value greater than or equal to 1. The default value is 1. The differential delay changes the shape, number, and location of nulls in the filter response. Increasing the differential delay increases the sharpness of the nulls and the response between the nulls. In practice, differential delay values of 1 or 2 are the most common.

Factor

> Specify the decimation or interpolation factor for your filter as an integer value greater than or equal to 1. The default value is 2.

Frequency units

> Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

> Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

> Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

> Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude units

> Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:
>
> • `Linear` — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Filter Implementation

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- Enforce single-rate processing — When you select this option, the block maintains the sample rate of the input.

- Allow multirate processing — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to Elements as channels (sample based).

Data Types Pane
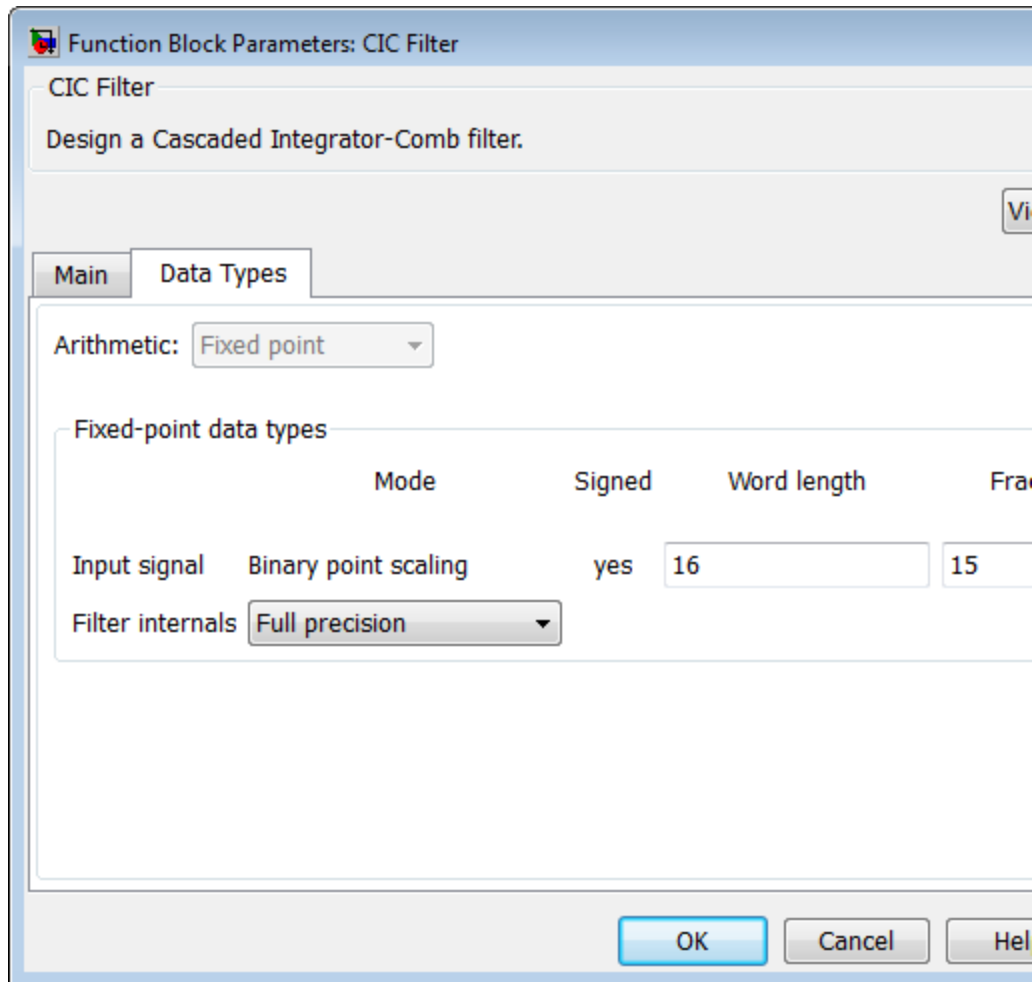
See the Data Types Pane subsection of the `filterbuilder` function reference page for more information about specifying data type parameters.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Fixed point

Purpose   Interpolate signal using Cascaded Integrator-Comb filter

Library   Filtering / Multirate Filters

dspmlti4
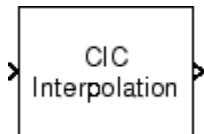
Description The CIC Interpolation block performs a sample rate increase
(interpolation) on an input signal by an integer factor. Cascaded
Integrator-Comb (CIC) filters are a class of linear phase FIR filters that
consist of a comb part and an integrator part.

The transfer function of a CIC interpolator filter is

$$H(z) = H_I{}^N(z)H_C{}^N(z) = \frac{(1-z^{-RM})^N}{(1-z^{-1})^N} = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N$$

where

- H_I is the transfer function of the integrator part of the filter.

- H_C is the transfer function of the comb part of the filter.

- N is the number of sections. The number of sections in a CIC filter
  is defined as the number of sections in either the comb part *or* the
  integrator part of the filter. This value does not represent the total
  number of sections throughout the entire filter.

- R is the interpolation factor.

- M is the differential delay.

The CIC Interpolation block requires a Simulink Fixed Point license.
The block supports real and complex fixed-point inputs and treats each
channel of a complex input as two real input channels.

CIC Filter Structure

The filter structures supported by the CIC Interpolation and CIC
Decimation blocks exactly match those created by the mfilt.cicinterp
and mfilt.cicdecim objects. You can create an mfilt object in any

workspace and specify that object in the **Multirate filter variable** parameter of this block. Alternatively, you can set the **Coefficient source** to **Dialog parameters** and specify information about the CIC filter using the available block dialog parameters.

You can use this block to create the following CIC filter structure. This interpolator has a latency of *N*, where *N* is the number of sections in either the comb or the integrator part of the filter.
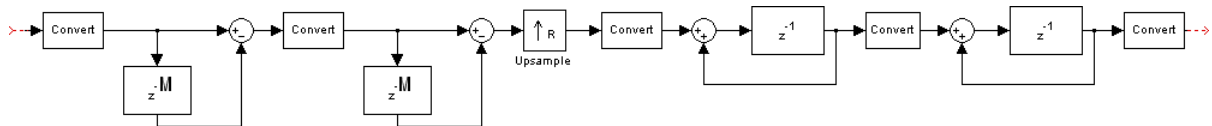
Dialog Box

Coefficient Source

The CIC Interpolation block can operate in two different modes. Select the mode in the **Coefficient source** group box. If you select:

- **Dialog parameters**, you enter information about the filter such as structure and coefficients in the block mask

- **Multirate filter object (MFILT)**, you specify the filter using an `mfilt` object.

Different items appear on the CIC Interpolation block dialog depending on whether you select **Dialog parameters** or **Multirate filter object (MFILT)** in the **Coefficient source** group box. See the following sections for details:

- "Specify Filter Characteristics in Dialog" on page 2-262

- "Specify Multirate Filter Object" on page 2-267

Specify Filter Characteristics in Dialog

The **Main** pane of the CIC Interpolation block dialog appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.
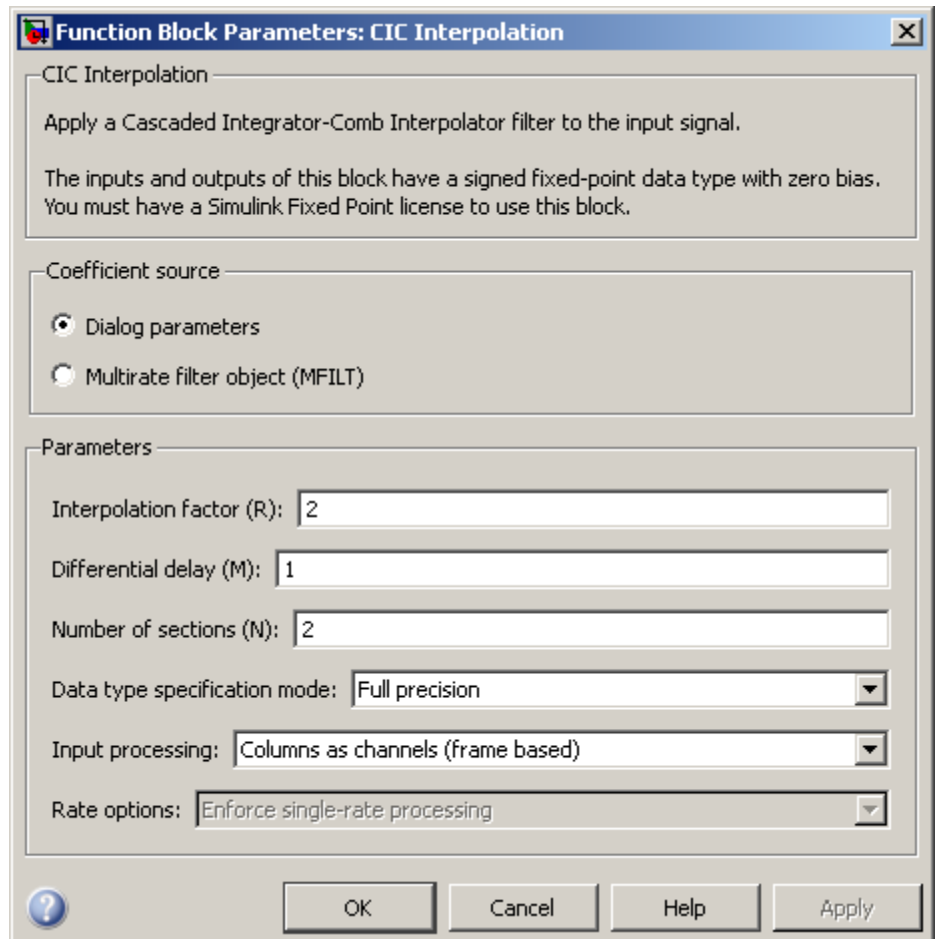
Interpolation factor (R)

Specify the interpolation factor of the filter.

Differential delay (M)

Specify the differential delay of the comb portion of the filter, M, as shown in the diagram in "CIC Filter Structure" on page 2-261.

Number of sections (N)

Specify the number of filter sections. The number you specify determines the number of sections in either the comb part of the filter or in the integrator part of the filter. This value does not represent the total number of sections in the comb and integrator parts combined.

Data type specification mode

Choose how you specify the fixed-point word length and fraction length of the filter sections and/or output.

- `Full precision` — In this mode, the word and fraction lengths of the filter sections and outputs are automatically selected for you. The output and last section word lengths are set to

$$\text{word length} = \text{ceil}(\log_2(\frac{(R*M)^N}{R})) + I$$

where

  — I = input word length

  — M = differential delay

  — N = number of sections

  — R = interpolation factor

The other section word lengths are set is such a way as to accommodate the bit growth, as described in Hogenauer's paper [1]. All fraction lengths are set to the input fraction length.

- `Minimum section word lengths` — In this mode, you specify the word length of the filter output in the **Output word length** parameter. The word lengths of the filter sections are set in the same way as in `Full precision` mode.

The section fraction lengths are set to the input fraction length. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.

- Specify word lengths — In this mode you specify the word lengths of the filter sections and output in the **Section word lengths** and **Output word length** parameters. The fraction lengths of the filter sections are set such that the spread between word length and fraction length is the same as in full-precision mode. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.

- Binary point scaling — In this mode you fully specify the word and fraction lengths of the filter sections and output in the **Section word lengths**, **Section fraction lengths**, **Output word length**, and **Output fraction length** parameters.

Section word lengths

Specify the word length, in bits, of the filter sections.

This parameter is only visible if you select Specify word lengths or Binary point scaling for the **Data type specification mode** parameter.

Section fraction lengths

Specify the fraction length of the filter sections.

This parameter is only visible if you select Binary point scaling for the **Data type specification mode** parameter.

Output word length

Specify the word length, in bits, of the filter output.

This parameter is only visible if you select Minimum section word lengths, Specify word lengths or Binary point scaling for the **Data type specification mode** parameter.

Output fraction length

Specify the fraction length of the filter output.

This parameter is only visible if you select Binary point scaling for the **Data type specification mode** parameter.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel. In this mode, the block always performs single-rate processing.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel. In this mode, the input to the block must be a scalar or a vector. You can use the **Rate options** parameter to specify whether the block performs single-rate or multirate processing.

Note The option Inherit from input (this choice will be removed - see release notes) will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

Specify the rate processing rule for the block. You can select one of the following options:

- Enforce single-rate processing — When you select this option, the block maintains the sample rate of the input.

- Allow multirate processing — When you select this option, the block produces an output with a sample rate that is R times faster than the input sample rate. To select this option, you must set the **Input processing** parameter to Elements as channels (sample based).

View filter response

This button opens the Filter Visualization Tool (fvtool) from the Signal Processing Toolbox product. The tool then displays

the filter response of the `mfilt` object specified in the **Multirate filter variable** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note   If you specify an `mfilt` object in the **Multirate filter variable** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

Specify Multirate Filter Object

The **Main** pane of the CIC Interpolation block dialog appears as follows when you specify **Multirate filter object (MFILT)** in the **Coefficient source** group box.

CIC Interpolation
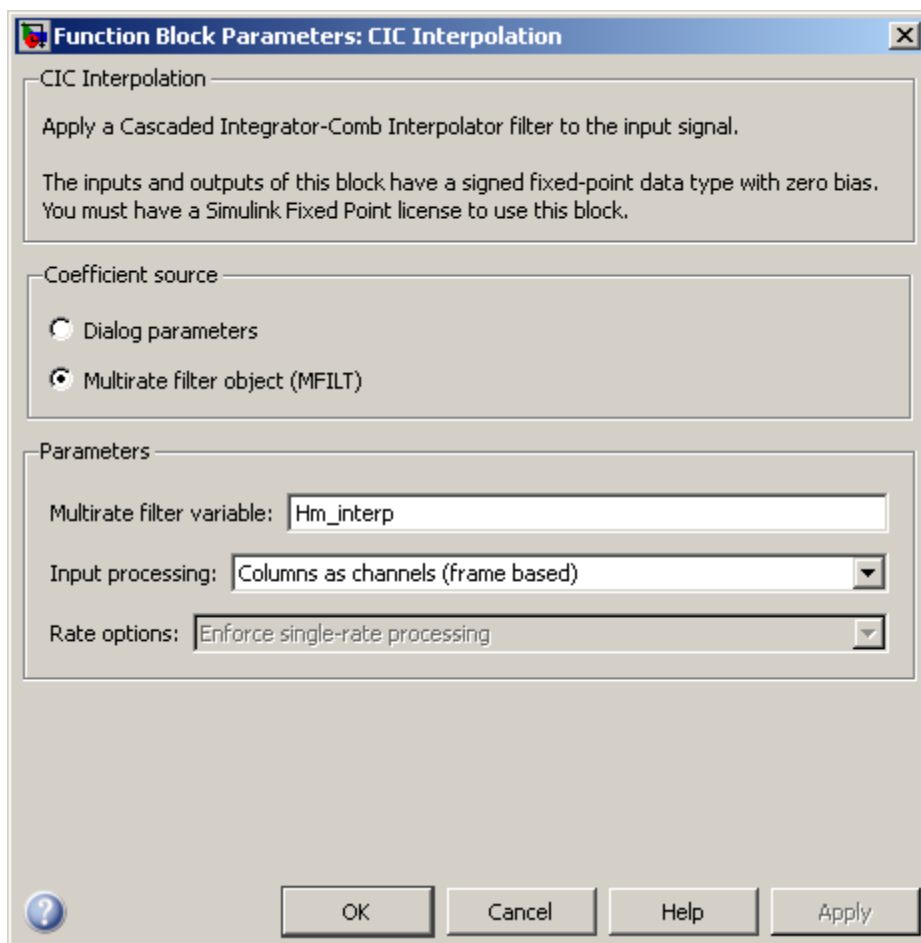
Multirate filter variable

Specify the multirate filter object (`mfilt`) that you want the block to implement. You can specify the `mfilt` object in one of three ways:

- You can fully specify the `mfilt` object in the block mask.

- You can enter the variable name of a `mfilt` object that is defined in any workspace.

- You can enter a variable name for a `mfilt` object that is not yet defined, as shown in the default value.

For more information on creating `mfilt` objects, see the `mfilt` function reference page.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel. In this mode, the block always performs single-rate processing.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. In this mode, the input to the block must be a scalar or a vector. You can use the **Rate options** parameter to specify whether the block performs single-rate or multirate processing.

Note The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

Specify the rate processing rule for the block. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.

CIC Interpolation

- Allow multirate processing — When you select this option, the block produces an output with a sample rate that is R times faster than the input sample rate. To select this option, you must set the **Input processing** parameter to Elements as channels (sample based).

View filter response
    This button opens the Filter Visualization Tool (fvtool) from the Signal Processing Toolbox product. The tool then displays the filter response of the mfilt object specified in the **Multirate filter variable** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note If you specify an mfilt object in the **Multirate filter variable** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

References   [1] Hogenauer, E.B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-29(2): pp. 155–162, 1981.

[2] Meyer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, Springer Verlag, 2001.

[3] Harris, Fredric J., *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

Supported Data Types
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

CIC Decimation	DSP System Toolbox
FIR Decimation	DSP System Toolbox

FIR Interpolation	DSP System Toolbox
filter	DSP System Toolbox
mfilt.cicdecim	DSP System Toolbox
mfilt.cicinterp	DSP System Toolbox

Comb Filter

Purpose   Design comb Filter

Library   Filtering / Filter Designs

dspfdesign

Description   This block brings the filter design capabilities of the filterbuilder
function to the Simulink environment.
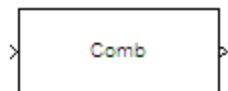
Dialog   See "Comb Filter Design Dialog Box—Main Pane" on page 7-703 for
Box   more information about the parameters of this block. The **Data Types**
and **Code Generation** panes are not available for blocks in the DSP
System Toolbox Filter Designs library.

Function Block Parameters: Comb Filter

Comb Filter

Design a comb filter.

View Filter Response

Filter specifications

Comb Type: Notch

Order mode: Order   Order: 10

Frequency specifications

Frequency constraints: Quality factor

Quality factor: 16

Frequency units: Normalized (0 to 1)   Input Fs: 2

Notch Frequencies: [0 0.2 0.4 0.6 0.8 1]

Magnitude specifications

No magnitude constraints can be specified when specifying a filter order.

Algorithm

Design method: Butterworth

Filter Implementation

Structure: Direct-form II

☐ Use basic elements to enable filter customization

Input processing: Columns as channels (frame based)

☐ Use symbolic names for coefficients

OK   Cancel   Help   Apply

Comb Filter

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify the type of comb filter and the number of peaks or notches.

Comb Type

Select either `Notch` or `Peak` from the drop-down list. `Notch` creates a comb filter that attenuates a set of harmonically related frequencies. `Peak` creates a comb filter that amplifies a set of harmonically related frequencies.

Order mode

Select either `Order` or `Number of Peaks/Notches` from the drop-down menu.

Select `Order` to enter the desired filter order in the dialog box. The comb filter has notches or peaks at increments of 2/`Order` in normalized frequency units.

Select `Number of Peaks` or `Number of Notches` to specify the number of peaks or notches and the `Shelving filter order`
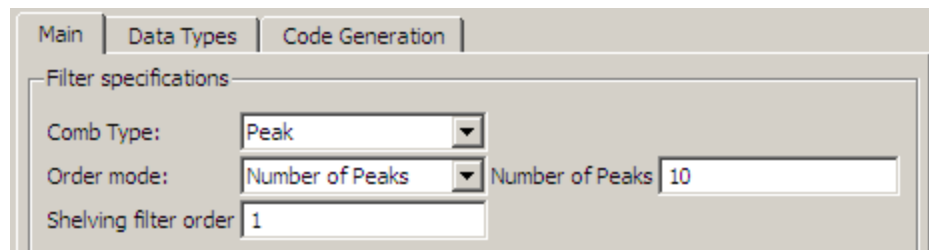
Shelving filter order

> The Shelving filter order is a positive integer that determines the sharpness of the peaks or notches. Larger values result in sharper peaks or notches.

Frequency specifications

In this group, you specify the frequency constraints and frequency units.

Frequency specifications

> Select either Quality factor or Bandwidth.
>
> Quality factor is the ratio of the center frequency of the peak or notch to the bandwidth calculated at the −3 dB point.
>
> Bandwidth specifies the bandwidth of the peak or notch. By default the bandwidth is measured at the −3 dB point. For example, setting the bandwidth equal to 0.1 results in 3 dB frequencies at normalized frequencies 0.05 above and below the center frequency of the peak or notch.

Frequency Units

> Specify the frequency units. The default is normalized frequency. Choosing an option in Hz enables the **Input Fs** dialog box.

Magnitude specifications

Specify the units for the magnitude specification and the gain at which the bandwidth is measured. This menu is disabled if you specify a filter order. Select one of the following magnitude units from the drop down list:

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Bandwidth gain — Specify the gain at which the bandwidth is measured. The default is –3 dB.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

The IIR Butterworth design is the only option for peaking or notching comb filters.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Note The Inherited (this choice will be removed see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- Enforce single-rate processing — When you select this option, the block maintains the sample rate of the input.

- Allow multirate processing — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to Elements as channels (sample based).

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

Complex Cepstrum

Purpose   Compute complex cepstrum of input

Library   Transforms

dspxfrm3

Description

The Complex Cepstrum block computes the complex cepstrum of each column in the real-valued *M*-by-*N* input matrix, u. The block treats each column of the input as an independent channel containing *M* consecutive samples. The block does not accept complex-valued inputs.

The input is altered by the application of a linear phase term so that there is no phase discontinuity at ±π radians. That is, each input channel is independently zero padded and circularly shifted to have zero phase at π radians.

The output is a real M_o-by-*N* matrix, where M_o is specified by the **FFT length** parameter. Each output column contains the length-M_o complex cepstrum of the corresponding input column.

```
y = cceps(u,Mo)     % Equivalent MATLAB code
```

When you select the **Inherit FFT length from input port dimensions** check box, the output frame size matches the input frame size (M_o=*M*). In this case, the block processes *sample-based* length-*M* row vector inputs as a single channel (that is, as an *M*-by-1 column vector), and returns the result as a length-*M* column vector. The block *always* processes unoriented vector inputs as a single channel, and returns the result as a length-*M* column vector.

The output port rate is the same as the input port rate.

Complex Cepstrum
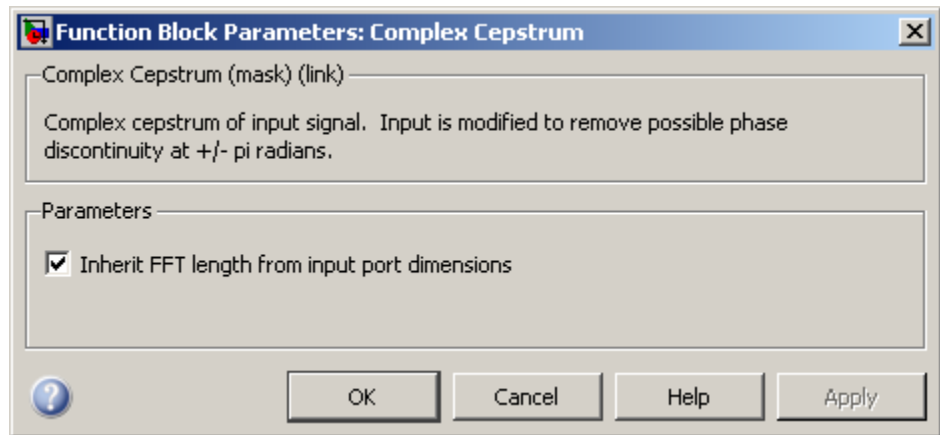
Dialog Box

Inherit FFT length from input port dimensions
> When you select this check box, the output frame size matches 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 visible only when you clear the **Inherit FFT length from input port dimensions** check box.

Supported Data Types

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

See Also

DCT	DSP System Toolbox
FFT	DSP System Toolbox
Real Cepstrum	DSP System Toolbox
cceps	Signal Processing Toolbox

Purpose   Compute complex exponential function

Library   Math Functions / Math Operations

dspmathops

Description   The Complex Exponential block computes the complex exponential function for each element of the real input, *u*.
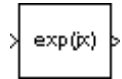
$$y = e^{ju} = \cos u + j \sin u$$

where $j = \sqrt{-1}$. The output is complex and the same size as the input.
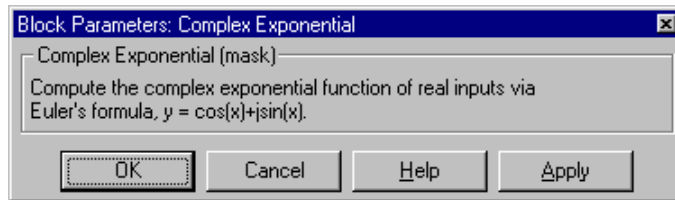
Dialog Box

Supported Data Types
- Double-precision floating point
- Single-precision floating point

See Also

Math Function	Simulink
Sine Wave	DSP System Toolbox
exp	MATLAB

Constant

Purpose	Generate constant value
Library	Sources
	dspsrcs4
Description	The Constant block is an implementation of the Simulink Constant block. See Constant for more information.

Purpose   Generate square, diagonal matrix

Library   • Sources

dspsrcs4

• Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

Description The Constant Diagonal Matrix block outputs a square diagonal matrix constant. The **Constant along diagonal** parameter determines the values along the matrix diagonal. This parameter can be a scalar to be repeated for all elements along the diagonal, or a vector containing the values of the diagonal elements. To generate the identity matrix, set the **Constant along diagonal** to 1, or use the Identity Matrix block.

The output is frame based when you select the **Frame-based output** check box; otherwise, the output is sample based.

Dialog Box The **Main** pane of the Constant Diagonal Matrix block dialog appears as follows.
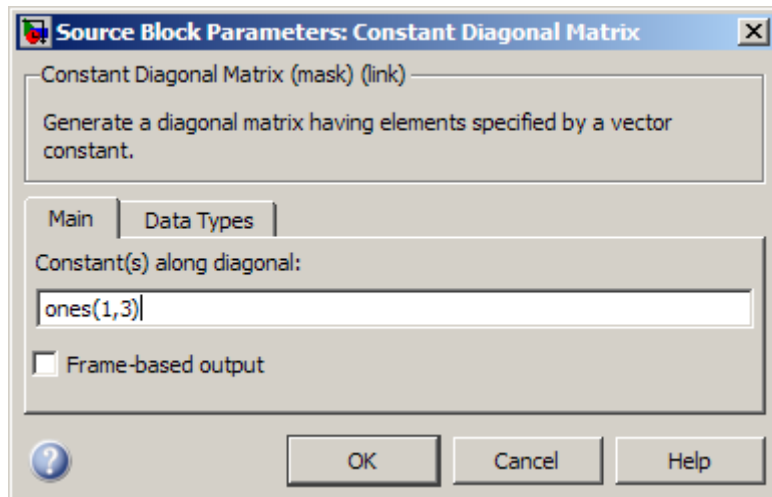
Constant Diagonal Matrix

Constant(s) along diagonal

> Specify the values of the elements along the diagonal. You can input a scalar or a vector. Tunable.
>
> When you specify any data type information in this field, it is overridden by the value of the **Output data type** parameter on the **Data Types** pane, unless you select `Inherit from 'Constant(s) along diagonal'`.

Frame-based output

> Select to cause the output of the block to be frame based. Otherwise, the output is sample based.

The **Data Types** pane of the Constant Diagonal Matrix block dialog appears as follows.
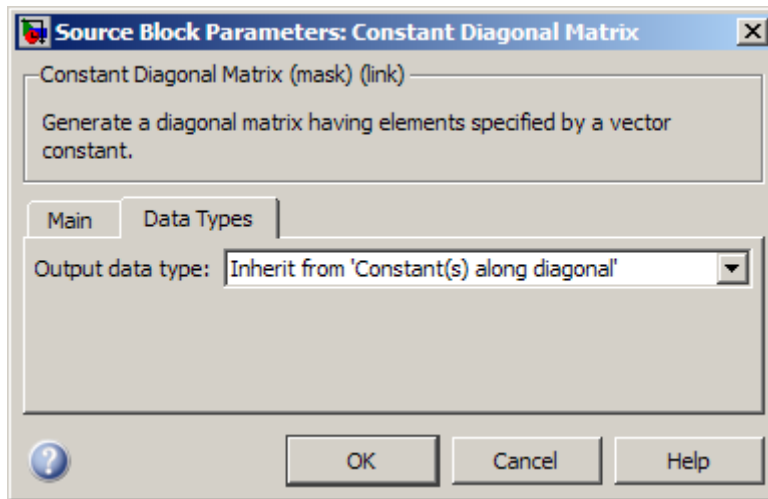
Output data type

> Specify the output data type in one of the following ways:
>
> • Choose one of the built-in data types from the list.

- Choose `Fixed-point` to specify the output data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `Inherit from 'Constant(s) along diagonal'` to set the output data type and scaling to match the values of the **Constant(s) along diagonal** parameter on the **Main** pane.

- Choose `Inherit via back propagation` to set the output data type and scaling to match the next block downstream.

The value of this parameter overrides any data type information specified in the **Constant(s) along diagonal** parameter on the **Main** pane, except when you select `Inherit from 'Constant(s) along diagonal'`.

Signed
Select to output a signed fixed-point signal. Otherwise, the signal is unsigned. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

Word length
Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type
Specify any built-in or fixed-point data type. You can specify fixed-point data types using the following Simulink Fixed Point functions: `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac`. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set fraction length in output to
Specify the scaling of the fixed-point output by either of the following two methods:

Constant Diagonal Matrix

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.

- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter, or when you select `User-defined` and the specified output data type is a fixed-point data type.

Fraction length
For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

Create Diagonal Matrix	DSP System Toolbox
Constant	Simulink
Identity Matrix	DSP System Toolbox
diag	MATLAB

Purpose   Generate ramp signal with length based on input dimensions

Library   Signal Operations

dspsigops

Description   The Constant Ramp block generates the constant ramp signal

```
y = (0:L-1)*m + b
```

where *m* is the slope specified by the scalar **Slope** parameter, and *b* is the *y*-intercept specified by the scalar **Offset** parameter.

For an unoriented vector input, *L* is equal to the length of the input vector. For an N-D input array, the length *L* of the output ramp is equal to the length of the input in the dimension specified by the **Ramp length equals number of** or **Dimension** parameter. The output, *y*, is always an unoriented vector.

Constant Ramp

Dialog Box

The **Main** pane of the Constant Ramp block dialog appears as follows.
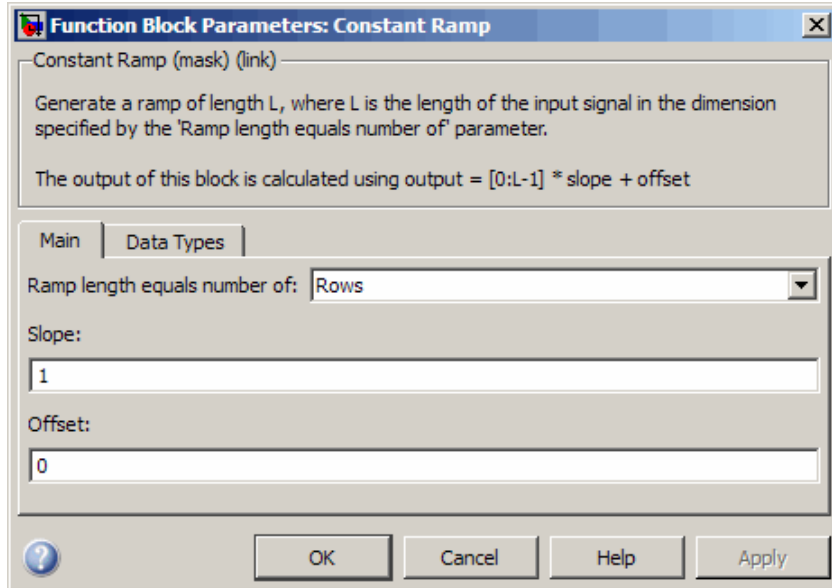
Ramp length equals number of
 Specify whether the length of the output ramp is the number of rows, number of columns, or the length of the specified dimension of the input.

Dimension
 Specify the one-based dimension of the input array that determines the length of the output ramp.

 This parameter is only visible when you select `Elements in specified dimension` for the **Ramp length equals number of** parameter.

Slope
 Specify the scalar slope of the ramp.

Offset

Specify the scalar *y*-intercept of the ramp.

The **Data Types** pane of the Constant Ramp block dialog appears as follows.
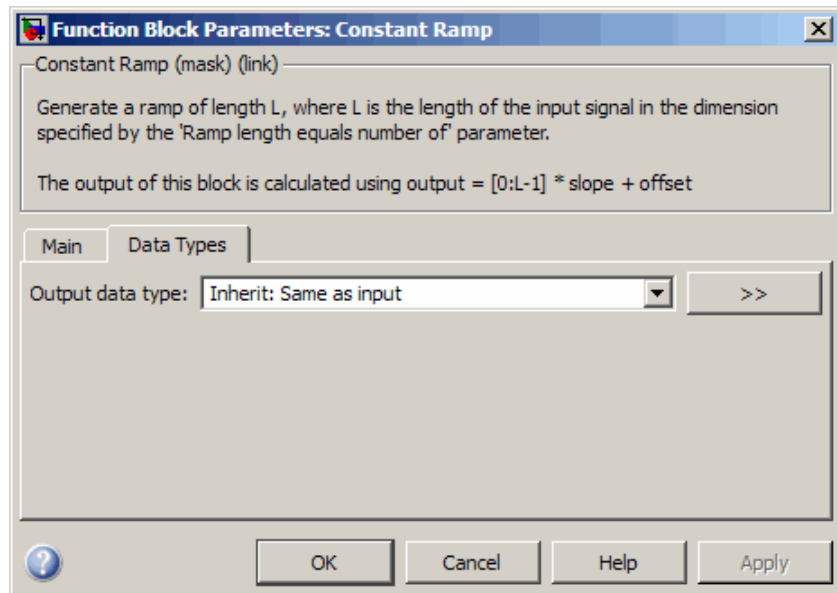
Output data type

Specify the output data type for this block. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Same as input`.

- A built in data type, such as `double`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Constant Ramp

Click the **Show data type assistant** button [>>] to
display the **Data Type Assistant**, which helps you set the
Output data type parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide*
for more information.

**Supported
Data
Types**

- Double-precision floating point

- Single-precision floating point

- Fixed point

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

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

This block differs from other DSP System Toolbox blocks in that unless
you choose `Same as input` for the **Output data type** parameter, the
data types of the input and the output do not need to be the same.

See Also

Create Diagonal Matrix	DSP System Toolbox
Constant	Simulink
Ramp	Simulink
Identity Matrix	DSP System Toolbox

Purpose   Create discontiguous input in contiguous block of memory

Library   dspobslib

Description
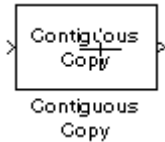
> **Note** The Contiguous Copy block is still supported but is likely to be obsoleted in a future release.

The Contiguous Copy block copies the input to a contiguous block of memory, and passes this new copy to the output. The output is identical to the input, but is guaranteed to reside in a contiguous section of memory.

Because Simulink software employs an efficient copy-by-reference method for propagating data in a model, some operations produce outputs with discontiguous memory locations.

Although this does not present a problem during simulation, blocks linked to versions of DSP Blockset prior to 4.0 may require contiguous inputs for code generation with the Simulink Coder product. When such blocks are used in a model intended for code generation, they should be preceded by the Contiguous Copy block to ensure that their inputs are contiguous.

Dialog Box
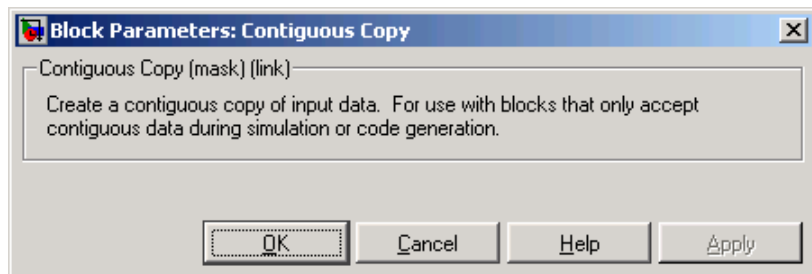
Contiguous Copy (Obsolete)

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

Purpose   Reshape 1-D or 2-D input to 2-D matrix with specified dimensions

Library   Signal Management / Signal Attributes

dspsigattribs

Description
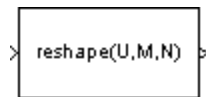
The Convert 1-D to 2-D block reshapes a length-M_i 1-D vector or an M_i-by-N_i matrix to an M_o-by-N_o matrix, where M_o is specified by the **Number of output rows** parameter, and N_o is specified by the **Number of output columns** parameter.

```
y = reshape(u,Mo,No)        % Equivalent MATLAB code
```

The input is reshaped *columnwise*, as shown in the two cases below. The length-6 vector and the 2-by-3 matrix are both reshaped to the same 3-by-2 output matrix.
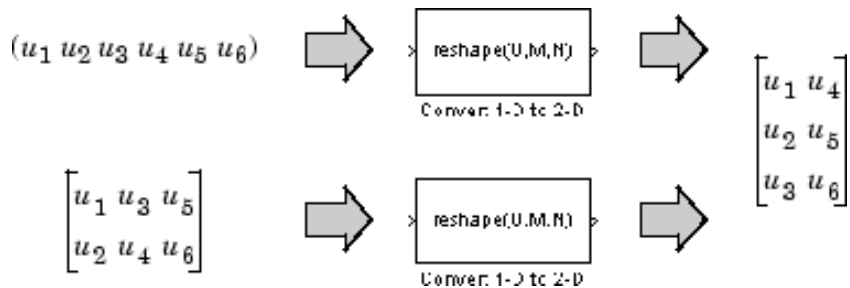
An error is generated when $(M_o*N_o)\neq(M_i*N_i)$. That is, the total number of input elements must be conserved in the output.

The output is frame based when you select the **Frame-based output** check box; otherwise, the output is sample based.

Convert 1-D to 2-D
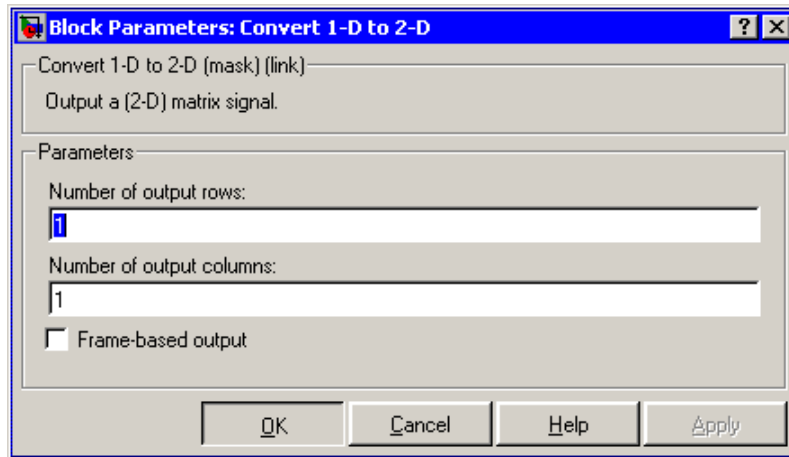
Number of output rows
    The number of rows, M_o, in the output matrix.

Number of output columns
    The number of rows, N_o, in the output matrix.

Frame-based output
    Creates a frame-based output when selected.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Buffer	DSP System Toolbox
Convert 2-D to 1-D	DSP System Toolbox
Frame Conversion	DSP System Toolbox
Reshape	Simulink
Submatrix	DSP System Toolbox

Convert 2-D to 1-D

Purpose   Convert 2-D matrix input to 1-D vector

Library   Signal Management / Signal Attributes

      dspsigattribs

Description The Convert 2-D to 1-D block reshapes an *M*-by-*N* matrix input to a 1-D
vector with length *M*N*.

```
y = u(:)      % Equivalent MATLAB code
```
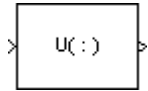
The input is reshaped *columnwise*, as shown below for a 3-by-2 matrix.

$$\begin{bmatrix} u_1 \; u_4 \\ u_2 \; u_5 \\ u_3 \; u_6 \end{bmatrix} \Longrightarrow \boxed{u(:)} \Longrightarrow (u_1 \; u_2 \; u_3 \; u_4 \; u_5 \; u_6)$$

**Dialog
Box**

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated

See Also

Buffer	DSP System Toolbox
Convert 1-D to 2-D	DSP System Toolbox
Frame Status Conversion (Obsolete)	DSP System Toolbox
Reshape	Simulink
Submatrix	DSP System Toolbox

Convolution

Purpose   Convolution of two inputs

Library   Signal Operations

dspsigops

Description   The Convolution block convolves the first dimension of an N-D input array u, with the first dimension of an N-D input array v. The block can also independently convolve a column vector with the first-dimension of an N-D input array.

Convolution with DSP System Toolbox Blocks

The general equation for convolution is

$$y(k) = \sum_n u(n-k) * h(k)$$

There are two DSP System Toolbox blocks that can be used for this purpose:

- Convolution
- Digital Filter

The Convolution block assumes that all of u and h are available at each Simulink time step, and computes the entire convolution at every one.

The Digital Filter block can be used for convolving signals in situations where all of h is available at each time step, but u is a sequence that comes in over the life of the simulation. When you use the Digital Filter block, the convolution is computed only once. To convolve inputs with the Digital Filter block, you must set the **Transfer function type** to FIR (all zeros).

Use the following questions to help you determine which block best fits your needs:

Question	Answer	Recommended Block(s)
How many convolutions do you intend to perform?	Many convolutions, one at each time step	• Convolution block
	One convolution over the life of the simulation	• Convolution block • Digital Filter block (in FIR mode)
How long are your input sequences?	Both sequences have a finite length	• Convolution block • Digital Filter block (in FIR mode)
	One sequence has an infinite (not predetermined) length	• Digital Filter block (in FIR mode)
How many of the inputs are scalar streams?	None	• Convolution block • Digital Filter block (in FIR mode)
	One or both	• Buffer block followed by the Convolution block • Digital Filter block (in FIR mode)

Convolving Two N-D Arrays

The block always computes the convolution of two N-D input arrays along the first dimension. When both inputs are N-D arrays, the size of their first dimension can differ, but the size of all other dimensions must be equal. For example, when u is an M_u-by-N-by-P array, and v is an M_v-by-N-by-P array, the output is an (M_u+M_v-1)-by-N-by-P array.

When the input to the Convolution block is a M_u-by-N matrix u and an M_v-by-N matrix v, the output, y, is a (M_u+M_v-1)-by-N matrix whose jth column has the following elements

Convolution

$$y_{i,j} = \sum_{k=0}^{\max(M_u,M_v)-1} u_{k,j} v_{(i-k),j} \quad 0 \le i \le (M_u + M_v - 2)$$

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.

Convolving a Column Vector with an N-D Array

When one input is a column vector and the other is an N-D array, the block independently convolves the vector with the first dimension of the N-D input array. For example, when u is a M_u-by-1 column vector and v is an M_v-by-N matrix, the output is an (M_u+M_v–1)-by-N matrix whose jth column has the following elements

$$y_{i,j} = \sum_{k=0}^{\max(M_u,M_v)-1} u_k v_{(i-k),j} \quad 0 \le i \le (M_u + M_v - 2)$$

Convolving Two Column Vectors

The Convolution block also accepts two column vector inputs. When u and v are column vectors with lengths M_u and M_v, the Convolution block performs the vector convolution

$$y_i = \sum_{k=0}^{\max(M_u,M_v)-1} u_k v_{(i-k)} \quad 0 \le i \le (M_u + M_v - 2)$$

The output is a (M_u+M_v–1)-by-1 column vector.

Fixed-Point Data Types

The following diagram shows the data types used within the Convolution block for fixed-point signals (time domain only).

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

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

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

Note When one or both of the inputs are signed fixed-point signals, all internal block data types are signed fixed point. The internal block data types are unsigned fixed point only when *both* inputs are unsigned fixed-point signals.

Dialog Box

The **Main** pane of the Convolution block dialog appears as follows.

Convolution

Computation domain

Set the domain in which the block computes convolutions:

- Time — The block computes in the time domain, which minimizes memory use.

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

- Fastest — The block computes in the domain, which minimizes the number of computations.

The **Data Types** pane of the Convolution block dialog appears as follows.

Convolution

Note Fixed-point signals are only supported for the time domain. To use the parameters on this pane, make sure Time is selected for the **Computation domain** parameter on the **Main** pane.

Rounding mode

Select the rounding mode for fixed-point operations.

Note The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is Inherit: Inherit via internal rule

- **Accumulator data type** is Inherit: Inherit via internal rule

- **Output data type** is Inherit: Same as accumulator

With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-300 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule

- An expression that evaluates to a valid data type, for example, fixdt([],16,0)

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-300 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-300 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum
> Specify the minimum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum
> Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
> Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Convolution

See Also

Correlation	DSP System Toolbox
Digital Filter	DSP System Toolbox
conv	MATLAB

Purpose   Cross-correlation of two inputs

Library   Statistics

dspstat3

Description   The Correlation block computes the cross-correlation of two N-D input
arrays. The block computes the correlation column-wise, so both inputs
must have the same number of columns. If one input is a column vector
and the other is an N-D array, the Correlation block computes the
cross-correlation of the vector with each column of the N-D array.

Correlating Two N-D Arrays

When the input to the Correlation block is an M_u-by-N input matrix u
and an M_v-by-N matrix v, the output, y, is a $(M_u + M_v - 1)$-by-N matrix
whose jth column has elements

$$y_{uv(i,j)} = \sum_{k=0}^{\max(M_u, M_v)-1} u_{k,j} v_{(k-i),j}^{*} \qquad 0 \le i < M_v$$

$$y_{uv(i,j)} = y_{vu(-i,j)}^{*} \qquad\qquad -M_u < i < 0$$

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.

Correlating a Column Vector with an N-D Array

When one input is a column vector and the other is an N-D array, the
Correlation block independently cross-correlates the input vector with
each column of the N-D array. For example, when u is a M_u-by-1 column
vector and v is an M_v-by-N matrix, the output is an $(M_u + M_v - 1)$-by-N
matrix whose jth column has elements

Correlation

$$y_{uv(i,j)} = \sum_{k=0}^{\max(M_u, M_v)-1} u_k v_{(k-i),j}^* \qquad 0 \le i < M_v$$

$$y_{uv(i,j)} = y_{vu(-i,j)}^* \qquad\qquad -M_u < i < 0$$

Correlating Two Column Vectors

The Correlation block also accepts two column vector inputs. When u and v are column vectors with lengths M_u and M_v, the Correlation block performs the vector cross-correlation according to the following equation:

$$y_{uv(i)} = \sum_{k=0}^{\max(M_u, M_v)-1} u_k v_{(k-i)}^* \qquad 0 \le i < M_v$$

$$y_{uv(i)} = y_{vu(-i)}^* \qquad\qquad -M_u < i < 0$$

The output is a $(M_u + M_v - 1)$-by-1 column vector.

Fixed-Point Data Types

The following diagram shows the data types used within the Correlation block for fixed-point signals (time domain only).

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

MULTIPLIER → CAST → ADDER (with $\frac{1}{z}$ feedback) → CAST

Input
data type(s)

Accumulator or
Product output
data type

Accumulator
data type

Accumulator
data type

Output data
type

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

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

Note When one or both of the inputs are signed fixed-point signals, all
internal block data types are signed fixed point. The internal block
data types are unsigned fixed point only when *both* inputs are unsigned
fixed-point signals.

Dialog Box

The **Main** pane of the Correlation block dialog appears as follows.

Correlation

Computation domain

Set the domain in which the block computes correlations:

- `Time` — The block computes in the time domain, which minimizes memory use.

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

- Fastest — The block computes in the domain, which minimizes the number of computations.

The **Data Types** pane of the Correlation block dialog appears as follows.

Correlation

Note Fixed-point signals are only supported for the time domain. To use the parameters on this pane, make sure `Time` is selected for the **Computation domain** parameter on the **Main** pane.

Rounding mode

Select the rounding mode for fixed-point operations.

> **Note** The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when all the following conditions exist:
>
> - **Product output data type** is `Inherit:  Inherit via internal rule`
>
> - **Accumulator data type** is `Inherit:  Inherit via internal rule`
>
> - **Output data type** is `Inherit:  Same as accumulator`
>
> With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-310 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

Correlation

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-310 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-310 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

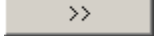Specify the minimum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Correlation

See Also

Autocorrelation	DSP System Toolbox
Convolution	DSP System Toolbox
`xcorr`	Signal Processing Toolbox

Purpose   Count up or down through specified range of numbers

Library   Signal Management / Switches and Counters

dspswit3
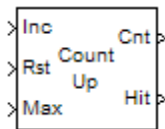
Description   The Counter block counts up or down through a specified range of numbers. The block enables the Inc (increment) port when you set the **Count direction** parameter to Up. When you set the **Count direction** parameter to Down, the block enables the Dec (decrement) port. If you set the **Count event** parameter to Free running, the block disables the Inc or Dec port, and counts at a constant time interval. For all other settings of the **Count event** parameter, the block increments or decrements the counter each time a trigger event occurs at the Inc or Dec input port. When a trigger event occurs at the optional Rst port, the block resets the counter to its initial state.

The Counter block accepts single-channel inputs. The Inc and Dec ports accept real-valued scalars or vectors. If the input to the Inc or Dec port is a vector, the block treats the vector as a frame. The Rst port only accepts real-valued scalars. The Rst port must have the same port sample time as the Inc or Dec input port. If you enable the optional Max input port, you must provide an unsigned integer input that the **Count data type** can represent.

See the following topics for more information:

- "Setting the Count Event Parameter" on page 2-320
- "Setting the Counter Size and Initial Count Parameters" on page 2-321
- "Scalar Input Operation" on page 2-322
- "Vector Input Operation" on page 2-323
- "Free-Running Operation" on page 2-324

Setting the Count Event Parameter

Specify the trigger event for the Inc/Dec and Rst ports by setting the **Count event** parameter to one of the following values:

- `Rising edge` — Triggers a count or reset operation when the input to the Inc/Dec or Rst port behaves in one of the following ways:

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

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure).
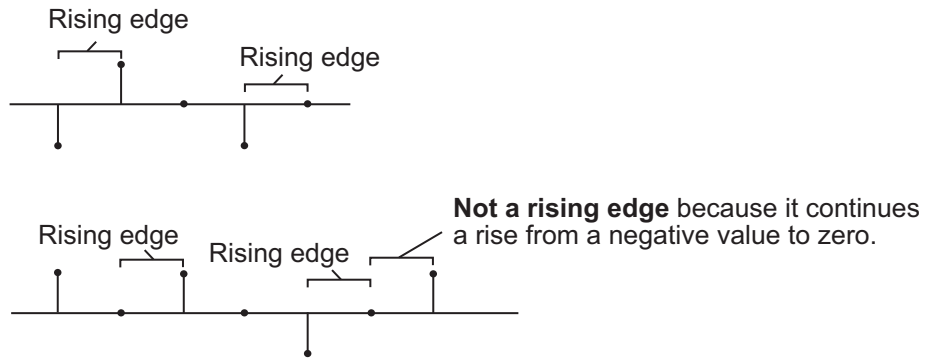
- `Falling edge` — Triggers a count or reset operation when the input to the Inc/Dec or Rst port behaves in one of the following ways:

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

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

Falling edge   Falling edge

Not a falling edge because it continues
a fall from a positive value to zero.

Falling edge

Falling edge
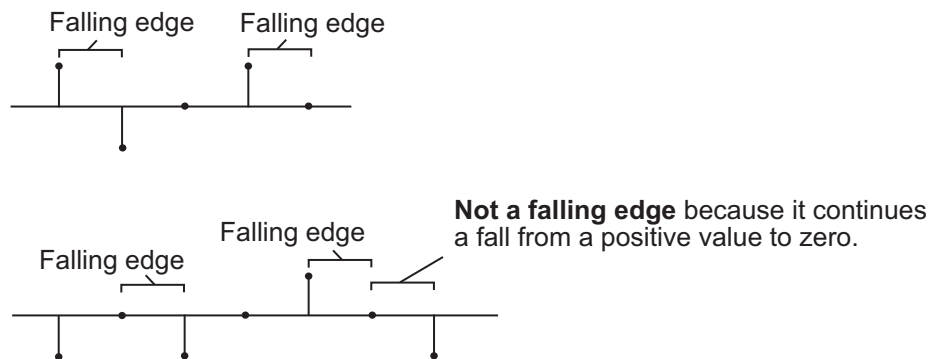
Falling edge

- `Either edge` — Triggers a count or reset operation when the input
  to the Inc/Dec or Rst port is a `Rising edge` or `Falling edge`.

- `Non-zero sample` — Triggers a count or reset operation at each
  sample time when the input to the Inc/Dec or Rst port is not zero.

- `Free running` — Disables the Inc/Dec port and enables the **Samples
  per output frame** and **Sample time** block parameters. The block
  increments or decrements the counter at a constant interval, T_s,
  which you specify using the **Sample time** parameter. For more
  information, see "Free-Running Operation" on page 2-324. In this
  mode, the block resets the counter whenever it receives a non-zero
  sample at the Rst port.

Setting the Counter Size and Initial Count Parameters

At the start of the simulation, the block sets the counter to the value you
specify in the **Initial count** parameter. The **Initial count** can be any
unsigned integer in the range defined by the **Counter size** parameter.

The **Counter size** parameter allows you to specify the range of integer
values the block counts through. When the block counts through the
entire counter range, the next time a trigger event occurs at the Inc/Dec
port, the block resets the counter as follows:

- When you set the **Count direction** parameter to `Up` and the counter
  reaches the upper-limit of the counter range, the block restarts the
  counter at zero.

Counter

- When you set the **Count direction** parameter to Down and the counter reaches zero, the block restarts the counter at the upper-limit of the counter range.

You can set the **Counter size** parameter to one of the following options:

- 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 **Maximum count** parameter, which allows you to specify the upper-count limit as any arbitrary unsigned integer that the **Count data type** can represent. The counter values range from 0 to the value of the **Maximum count** parameter.

- Specify via input port — Enables the Max input port, which allows you to specify the upper-count limit as any arbitrary unsigned integer that the **Count data type** can represent. The counter values range from 0 to the value you specify as an input to the Max port.

Scalar Input Operation

When you set the **Count direction** parameter to Up, a trigger event at the Inc (increment) input port causes the block to increase the counter by one. Assuming no reset events occur, the block continues increasing the counter value when triggered, until the counter value reaches the upper-count limit. The next time a trigger event occurs at the Inc port, the block restarts the counter at 0 and resumes increasing the counter by one for each subsequent trigger event at the Inc port.

When you set the **Count direction** parameter to Down, a trigger event at the Dec (decrement) input port causes the block to decrease the counter by one. Assuming no reset events occur, the block continues decreasing the counter value when triggered until the counter value reaches zero. The next time a trigger event occurs at the Dec port, the block restarts the counter at the upper-count limit and resumes decreasing the counter by one for each subsequent trigger event at the Dec port.

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 by the **Count event** parameter occurs at the optional Rst input port. When the Inc/Dec and Rst ports receive trigger events simultaneously, the block first resets the counter, and then increments or decrements the counter appropriately. If you do not need to reset the counter during simulation, you can disable the Rst port by clearing the **Reset input** check box.

The **Output** parameter allows you to specify which values the block outputs.

- Count enables a Cnt output port on the block, which provides the current value of the counter as a scalar value. The Cnt output port has the same port sample time as the Inc/Dec input port.

- Hit enables a Hit output port on the block. The Hit port produces zeros while the value of the counter does not equal any of the integers you specify for the **Hit values** parameter. You can specify an integer or a vector of integers for the **Hit values** parameter. When the counter value does equal one or more of the values you specify for the **Hit values** parameter, the block outputs a value of 1 at the Hit output port. The Hit output port has the same port sample time as the Inc/Dec input port.

- Count and Hit enables both the Cnt and Hit output ports.

Vector Input Operation

The block treats vector inputs to the Inc/Dec port as a frame. Vector operation is the same as scalar operation, except that the block increments or decrements the counter by the total number of trigger events contained in the Inc/Dec input vector. Thus, the counter may change multiple times during the processing of a single Inc/Dec input vector.

When the block has a Hit port, the block outputs a value of 1 if any of the **Hit values** match any of the counter values during the processing of the Inc/Dec input vector.

Counter

When a trigger event splits across two consecutive vectors, that event is counted in the vector that contains the conclusion of the event. When the Rst port receives a trigger event at the same time as the Inc/Dec port, the block first resets the counter. The block then increments or decrements the counter by the number of trigger events contained in the Inc/Dec input vector.

When the input to the Inc/Dec port is a length *N* vector, the port sample time of the Inc/Dec input port is equal to the frame period of the input, or *N* times the sample time of the input signal. The port sample time of the Cnt and Hit output ports equals that of the Inc/Dec input port.

Free-Running Operation

The block operates in free-running mode when you select `Free running` for the **Count event** parameter.

The Inc/Dec input port is disabled in this mode, and the block simply increments or decrements the counter at the constant interval, T_s, which you specify using the **Sample time** parameter.

In this mode, the Rst port always behaves as if the **Count event** parameter were set to `Non-zero sample`. Thus, the block triggers a reset event at each sample time that the Rst input is not zero.

In this mode, the Cnt output is an *M*-by-1 vector containing the count value at each of *M* consecutive sample times, where *M* is the value you specify for the **Samples per output frame** parameter. The Hit output is an *M*-by-1 vector containing the hit status (0 or 1) at each of those *M* consecutive sample times. Both the Cnt and Hit output ports have a port sample time of $M*T_s$.

Examples
In the following model, ex_counter_ref, the Simulink Pulse Generator block drives the Dec port of the Counter block, and the N-Sample Enable block triggers the Rst port. All inputs to and outputs from the Counter block are multiplexed into a single To Workspace block using a 4-port Mux block.
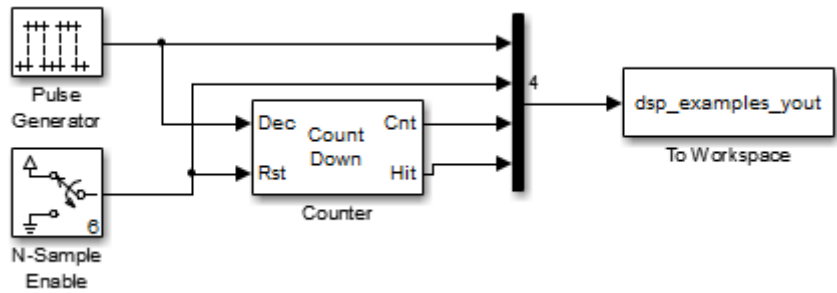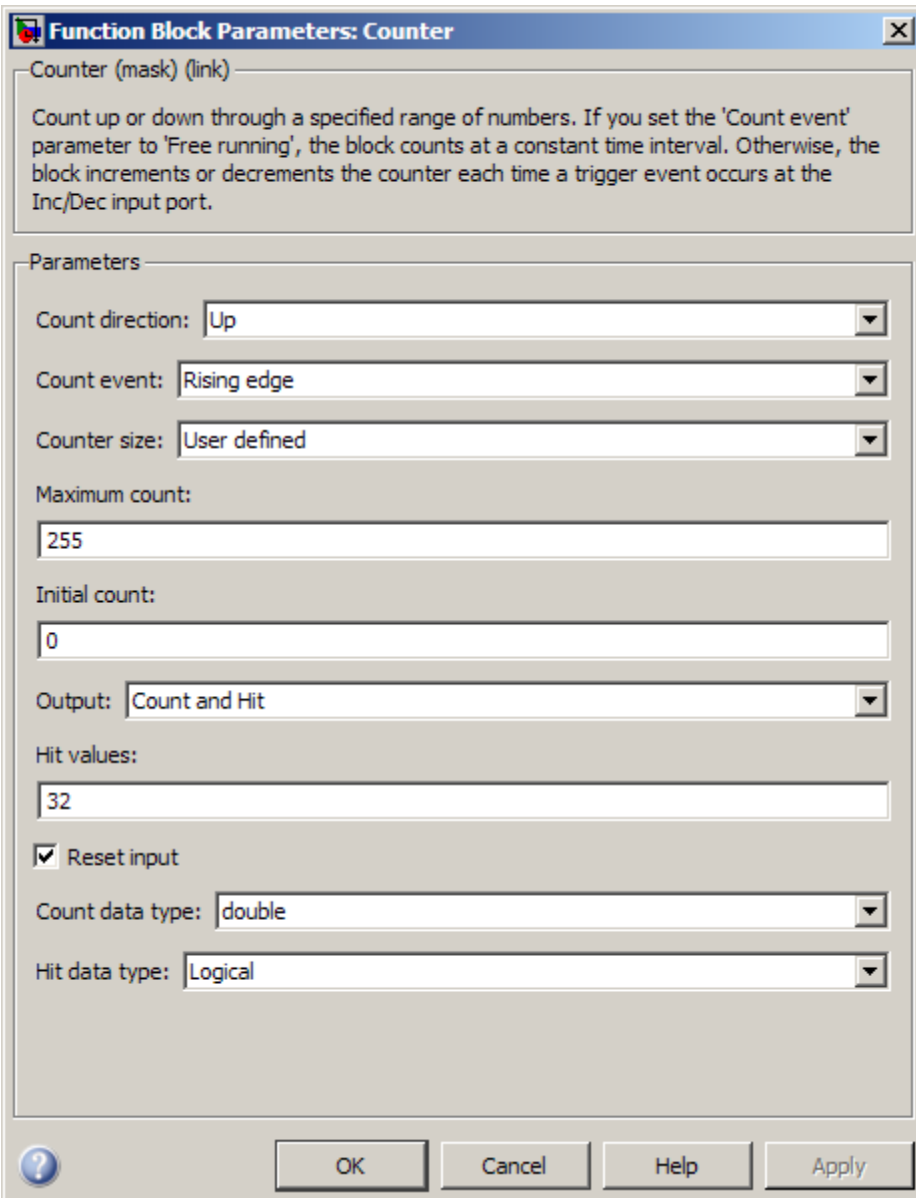
The following figure shows the first 22 samples of the model's four-column output, yout.

Counter
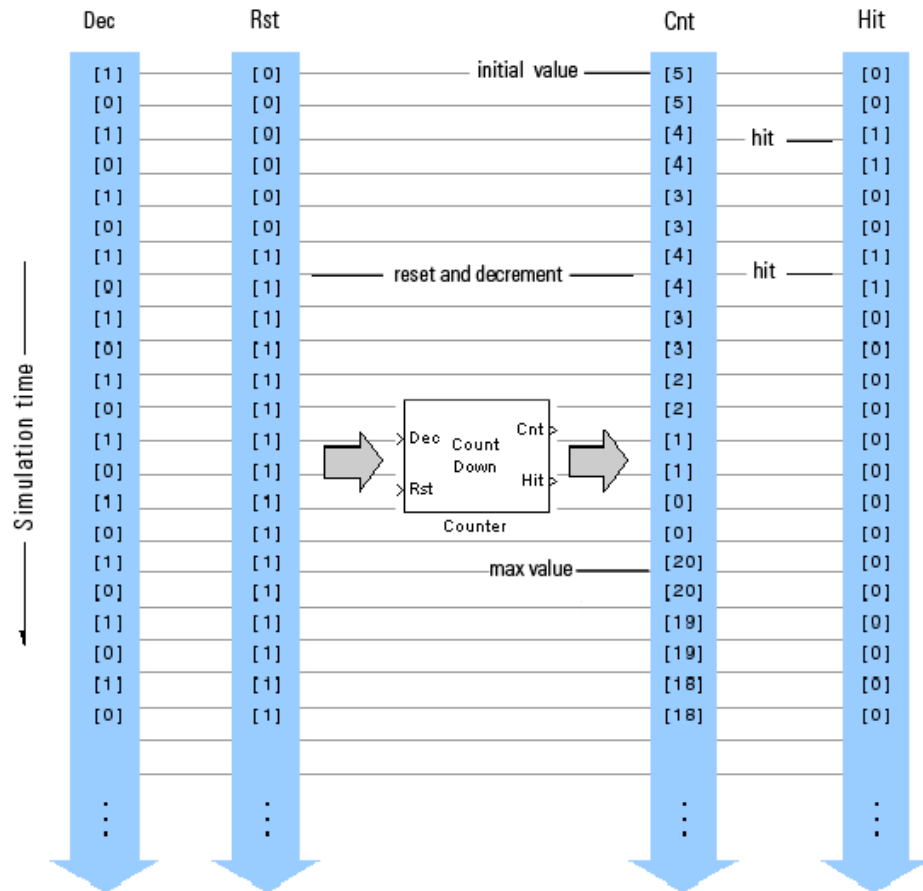
You can see that the seventh input sample to both the Dec and Rst ports
of the Counter block represent trigger events (rising edges). When this
occurs, the block first resets the counter to its initial value of 5, and then
immediately decrements the counter to 4. When the counter reaches
its minimum value of 0, the block restarts the counter at its maximum
value of 20 the next time a trigger event occurs at the Dec port.

Function Block Parameters: Counter

Counter (mask) (link)

Count up or down through a specified range of numbers. If you set the 'Count event' parameter to 'Free running', the block counts at a constant time interval. Otherwise, the block increments or decrements the counter each time a trigger event occurs at the Inc/Dec input port.

Parameters

Count direction: Up

Count event: Rising edge

Counter size: User defined

Maximum count:

255

Initial count:

0

Output: Count and Hit

Hit values:

32

☑ Reset input

Count data type: double

Hit data type: Logical

OK   Cancel   Help   Apply

Dialog Box

Count direction

Specify whether to count Up or Down. The port label on the block icon changes to Inc (increment) or Dec (decrement) based on the value of this parameter.

- When you set the **Count direction** parameter to Up and the counter reaches the upper-limit of the counter range, the block

Counter

restarts the counter at zero the next time a trigger event occurs at the Inc port.

- When you set the **Count direction** parameter to Down and the counter reaches zero, the block restarts the counter at the upper-limit of the counter range the next time a trigger event occurs at the Dec port.

This parameter is tunable in Simulink Normal mode.

Count event
Specify the type of event that triggers the block to increment, decrement, or reset the counter when received at the Inc/Dec or Rst ports. When you set this parameter to Free running, the block disables the Inc/Dec port and counts at the constant interval specified by the **Sample time** parameter. For more information on all the possible settings of this parameter, see "Setting the Count Event Parameter" on page 2-320.

Counter size
Specify the range of integer values the block counts through. For more information about the valid values of this parameter, see "Setting the Counter Size and Initial Count Parameters" on page 2-321.

Maximum count
Specify the maximum value of the counter as any unsigned integer representable by the data type you specify for the **Counter data type** parameter. This parameter appears only when you set the **Counter size** parameter to User defined. Tunable in Simulink Normal mode.

Initial count
Specify the initial value of the counter. The block uses the initial value of the counter at the start of simulation and resets the counter back to that initial value each time a trigger event occurs at the Rst port. Tunable.

Output

Select the output ports to enable. You can choose to enable the Count, Hit, or Count and Hit ports.

Hit values

Specify an integer or vector of integers whose occurrence in the count should be flagged by a 1 at the (optional) Hit output port. This parameter appears only when you set the **Output** parameter to Hit or Count and Hit. Tunable.

Reset input

Select this check box to enable the Rst input port. When you enable the Rst port, the block resets the counter to its initial value each time a trigger event occurs at the Rst port. To specify the type of event that triggers a reset of the counter, set the **Count event** parameter. When you clear the **Reset input** check box, you cannot reset the counter during simulation.

Samples per output frame

Specify the number of samples, *M*, in each output vector. This parameter appears only when you set the **Count event** parameter to Free running.

Sample time

Specify the constant interval, T_s, at which the block increments or decrements the counter when in free-running mode. For example, to have the block increment the counter every 5 seconds, set the **Count direction** parameter to Up, the **Count event** parameter to Free running, and specify a value of 5 for the **Sample time** parameter. In free running mode, the sample time of the output ports is always $M*T_s$.

This parameter appears only when you set the **Count event** parameter to Free running.

Count data type

Specify the data type of the output at the Cnt port. This parameter appears only when you set the **Output** parameter to Count or Count and Hit.

Hit data type
 Specify the data type of the output at the Hit port. This parameter appears only when you set the **Output** parameter to Hit or set it to Count and Hit with the **Count data type** parameter set to Double.

Supported Data Types

Port	Supported Data Types
Inc/Dec	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Rst	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Max	• 8-, 16-, and 32-bit unsigned integers
Cnt	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Hit	• Logical • Boolean — The block might output Boolean values from the Hit output port depending on the setting of the **Hit data type** parameter.

See Also

Edge Detector	DSP System Toolbox
N-Sample Enable	DSP System Toolbox
N-Sample Switch	DSP System Toolbox

Covariance AR Estimator

Purpose   Compute estimate of autoregressive (AR) model parameters using covariance method

Library   Estimation / Parametric Estimation

dspparest3

Description   The Covariance AR Estimator block uses the covariance method to fit an autoregressive (AR) model to the input data. This method minimizes the forward prediction error in the least squares sense.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal, which is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input frame.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

The order, p, of the all-pole model is specified by the **Estimation order** parameter. To guarantee a valid output, you must set the **Estimation order** parameter to be less than or equal to half the input vector length.

The top output, A, is a column vector of length $p+1$ with the same frame status as the input, and contains the normalized estimate of the AR model coefficients in descending powers of z.

[1 a(2) ... a(p+1)]

The scalar gain, G, is provided at the bottom output (G).

See the Burg AR Estimator block reference page for a comparison of the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

Dialog Box

Block Parameters: Covariance AR Estimator ⊠

Covariance AR Estimator (mask)

Frame-based parametric AR estimation using the Covariance method. The AR model coefficients are given in A and the gain is given in G.

Parameters

Estimation order:

4

| OK | Cancel | Help | Apply |
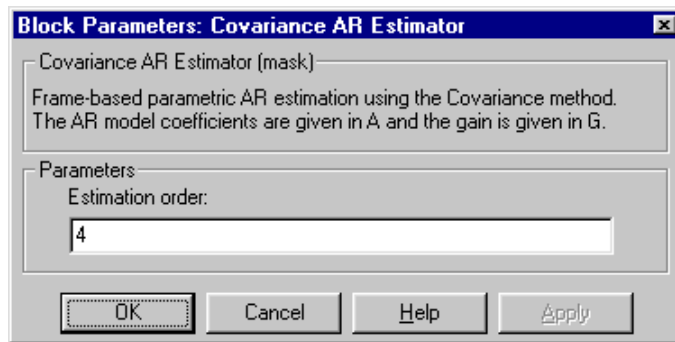
Estimation order

The order of the AR model, *p*. To guarantee a nonsingular output, you must set *p* to be less than or equal to half the input length. Otherwise, the output might be singular.

References Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
A	• Double-precision floating point
	• Single-precision floating point
G	• Double-precision floating point
	• Single-precision floating point

Covariance AR Estimator

See Also

Burg AR Estimator	DSP System Toolbox
Covariance Method	DSP System Toolbox
Modified Covariance AR Estimator	DSP System Toolbox
Yule-Walker AR Estimator	DSP System Toolbox
arcov	Signal Processing Toolbox

Purpose   Power spectral density estimate using covariance method

Library   Estimation / Power Spectrum Estimation

dspspect3

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 **Estimation order** parameter specifies the order of the all-pole model. The block computes the spectrum from the FFT of the estimated AR model parameters. To guarantee a valid output, the **Estimation order** parameter must be less than or equal to half the input vector length.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only). It represents a frame of consecutive time samples from a single-channel signal. The block outputs a column vector containing the estimate of the power spectral density of the signal at N_{fft} equally spaced frequency points. The frequency points are in the range $[0,F_s)$, where F_s is the sampling frequency of the signal.

Selecting **Inherit FFT length from estimation order**, specifies that N_{fft} is one greater than the estimation order. Clearing the **Inherit FFT length from estimation order** check box allows you to use the **FFT length** parameter to specify N_{fft} as a power of 2. The block zero-pads or wraps the input to N_{fft} before computing the FFT. The output is always sample based.

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must 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 equals the sample period of the original time series.

Covariance Method

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

See the Burg Method block reference for a comparison of the Burg Method, Covariance Method, Modified Covariance Method, and Yule-Walker Method blocks.
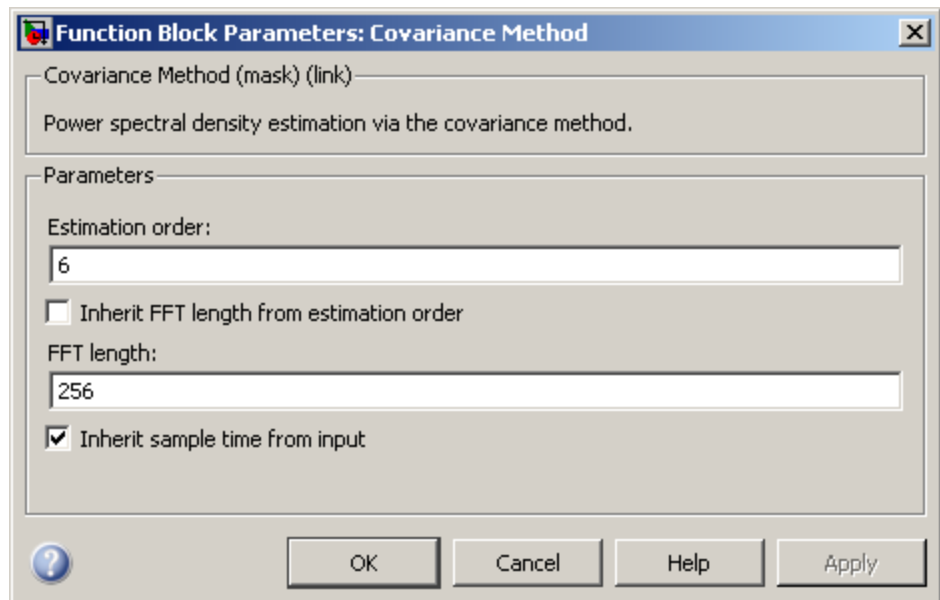
Dialog Box

Estimation order

The order of the AR model. To guarantee a nonsingular output, the value of this parameter must be less than or equal to half the input length.

Inherit FFT length from estimation order

When selected, this option specifies that the FFT length is one greater than the estimation order.

FFT length

Enter the number of data points on which to perform the FFT, N_{fft}. When N_{fft} is larger than the input frame size, the block zero-pads each frame as needed. When N_{fft} is smaller than the input frame size, the block wraps each frame as needed. This parameter becomes visible only when you clear the **Inherit FFT length from estimation order** check box.

Inherit sample time from input

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must 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 equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

Sample time of original time series

Specify the sample time of the original time-domain signal. This parameter becomes visible only when you clear the **Inherit sample time from input** check box.

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.

Orfanidis, S. J. *Introduction to Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1995.

Covariance Method

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

See Also

Burg Method	DSP System Toolbox
Covariance AR Estimator	DSP System Toolbox
Modified Covariance Method	DSP System Toolbox
Short-Time FFT	DSP System Toolbox
Yule-Walker Method	DSP System Toolbox
spectrum.cov	Signal Processing Toolbox

See "Spectral Analysis" for related information.

Purpose   Create square diagonal matrix from diagonal elements

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3
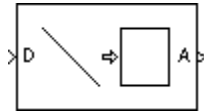
Description   The Create Diagonal Matrix block populates the diagonal of the M-by-M matrix output with the elements contained in the length-M vector input, D. The elements off the diagonal are zero.

```
A = diag(D)      Equivalent MATLAB code
```


Dialog Box
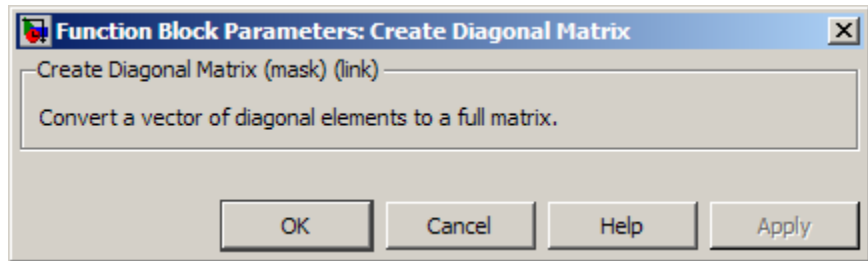
Supported Data Types

Port	Supported Data Types
D	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Create Diagonal Matrix

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Constant Diagonal Matrix	DSP System Toolbox
Extract Diagonal	DSP System Toolbox
diag	MATLAB

Purpose   Cumulative product of channel, column, or row elements

Library   Math Functions / Math Operations

dspmathops

Description   The Cumulative Product block computes the cumulative product along the specified dimension of the input or across time (running product).

The input can be a vector or matrix. The output always has the same dimensions, rate, data type, and complexity as the input.

Input and Output Characteristics

Valid Input
The Cumulative Product block accepts vector or matrix inputs containing real or complex values.

Valid Reset Signal
The optional reset port, Rst, accepts scalar values, which can be any built-in Simulink data type including boolean. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time.

Output Characteristics
The output always has the same dimensions, rate, data type, and complexity as the data signal input.

Computing the Running Product Along Channels of the Input

When you set the **Multiply input along** parameter to Channels (running product), the block computes the cumulative product of the elements in each input channel. The running product of the current input takes into account the running product of all previous inputs. In this mode, you must also specify a value for the **Input processing** parameter. When you set the **Input processing** parameter to Columns as channels (frame based), the block computes the running product along each column of the current input. When you set the **Input processing** parameter to Elements as channels (sample based),

Cumulative Product

the block computes a running product for each element of the input across time. See the following sections for more information:

- Computing the Running Product for Each Column of the Input on page 342

- Computing the Running Product for Each Element of the Input on page 343

- Resetting the Running Product on page 344

Computing the Running Product for Each Column of the Input

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first row of the first output is the same as the first row of the first input.

- The first row of each subsequent output is the element-wise product of the first row of the current input (time t), and the last row of the previous output (time t - T_f, where T_f is the frame period).

- The output has the same size, dimension, data type, and complexity as the input.

Given an M-by-N matrix input, u, the output, y, is an M-by-N matrix whose first row has elements

$$y_{1,j}(t) = u_{1,j}(t) \cdot y_{M,j}(t - T_f)$$

Running Product for Each Column of the Input

Computing the Running Product for Each Element of the Input

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the input matrix as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first output is the same as the first input.

- Each subsequent output is the element-wise product of the current input (time t) and the previous output (time t - T_s, where T_s is the sample period).

- The output has the same size, dimension, data type, and complexity as the input.

Given an M-by-N matrix input, u, the output, y, is an M-by-N matrix with the elements

$$y_{i,j}(t) = u_{i,j}(t) \cdot y_{i,j}(t - T_s) \quad \begin{array}{l} 1 \le i \le M \\ 1 \le j \le N \end{array}$$

Cumulative Product

For convenience, the block treats length-M unoriented vector inputs as M-by-1 column vectors when multiplying along channels. In such cases, the output is a length-M unoriented vector.

Running Product for Each Element of the Input
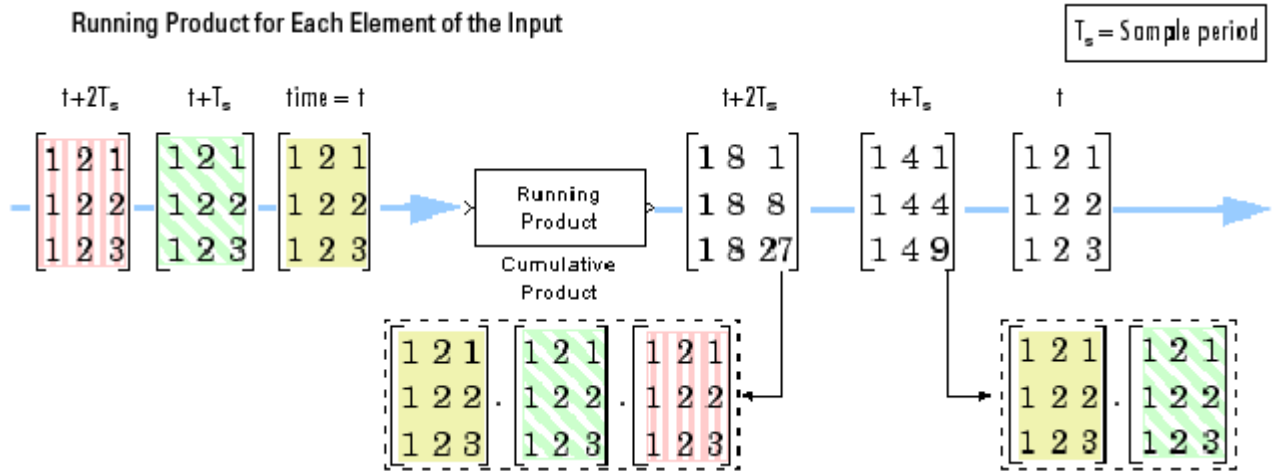
Resetting the Running Product

When you are computing the running product, you can configure the block to reset the running product whenever it detects a reset event at the optional Rst port. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. The input to the Rst port can be of the Boolean data type.

If a reset event occurs while the block is performing sample-based processing, the block initializes the current output to the values of the current input. If a reset event occurs while the block is performing frame-based processing, the block initializes the first row of the current output to the values in the first row of the current input.

The **Reset port** parameter specifies the reset event, which can be one of the following:

• None disables the Rst port.

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

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

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

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

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

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

Cumulative Product
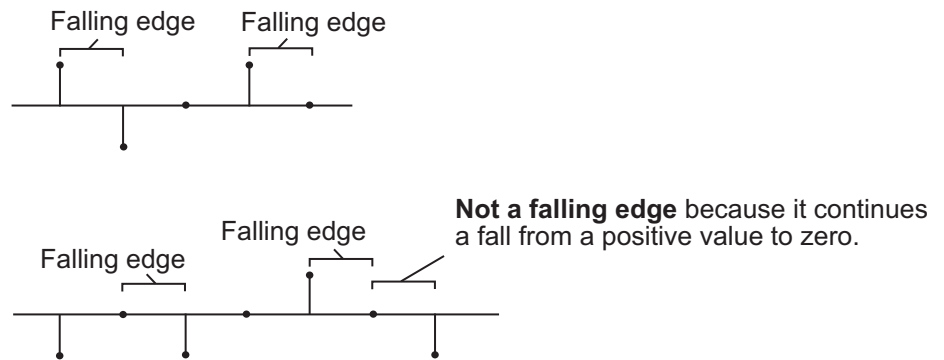
Falling edge   Falling edge

Falling edge

Falling edge

Not a falling edge because it continues a fall from a positive value to zero.

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

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

---

Note When you run simulations in Simulink `MultiTasking` mode, reset signals have a one-sample latency. Thus, when the block detects a reset event, a one-sample delay occurs at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

Multiplying Along Columns

When you set the **Multiply input along** parameter to `Columns`, the block computes the cumulative product of each column of the input. In this mode, the current cumulative product is independent of the cumulative products of previous inputs.

```
y = cumprod(u)      % Equivalent MATLAB code
```

The output has the same size, dimension, data type, and complexity as the input. The mth output row is the element-wise product of the first m input rows.

Given an M-by-N input, u, the output, y, is an M-by-N matrix whose jth column has elements

$$y_{i,j} = \prod_{k=1}^{i} u_{k,j} \qquad 1 \le i \le M$$

When multiplying along columns, the block treats length-M unoriented vector inputs as M-by-1 column vectors.

Product Along Columns
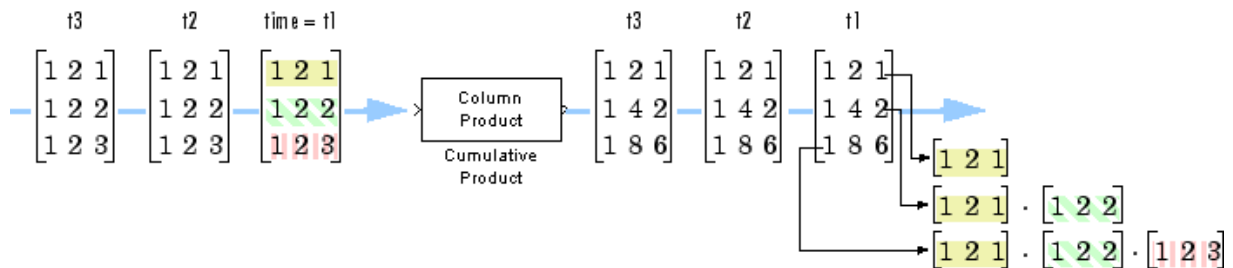
Multiplying Along Rows

When you set the **Multiply input along** parameter to Rows, the block computes the cumulative product of the row elements. In this mode, the current cumulative product is independent of the cumulative products of previous inputs.

```
y = cumprod(u,2)      % Equivalent MATLAB code
```

The output has the same size, dimension, and data type as the input. The nth output column is the element-wise product of the first n input columns.

Given an M-by-N matrix input, u, the output, y, is an M-by-N matrix whose ith row has elements

Cumulative Product

$$y_{i,j} = \prod_{k=1}^{j} u_{i,k} \qquad 1 \le j \le N$$

When you multiply along rows, the block treats length-N unoriented vector inputs as 1-by-N row vectors.

Product Along Rows
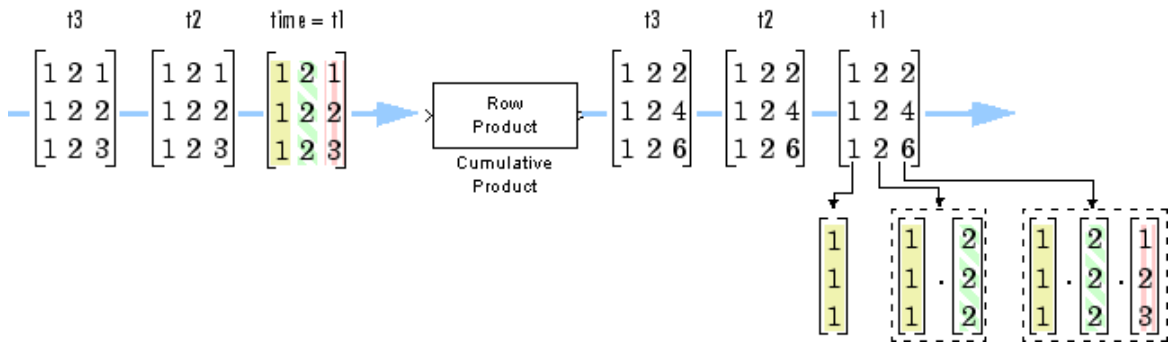
Fixed-Point Data Types

The following diagram shows the data types used within the Cumulative Product block for fixed-point signals.
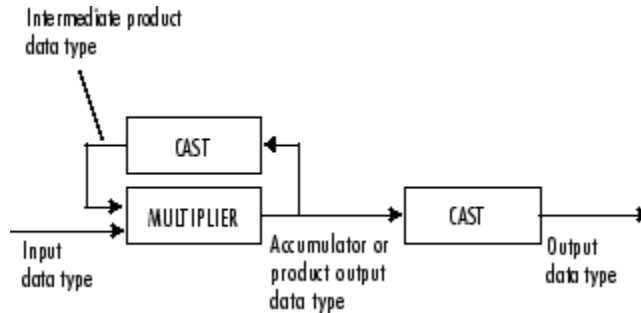
The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is

in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types". You can set the accumulator, product output, intermediate product, and output data types in the block dialog as discussed in "Dialog Box" on page 2-349.

Dialog Box

The **Main** pane of the Cumulative Product block dialog appears as follows.
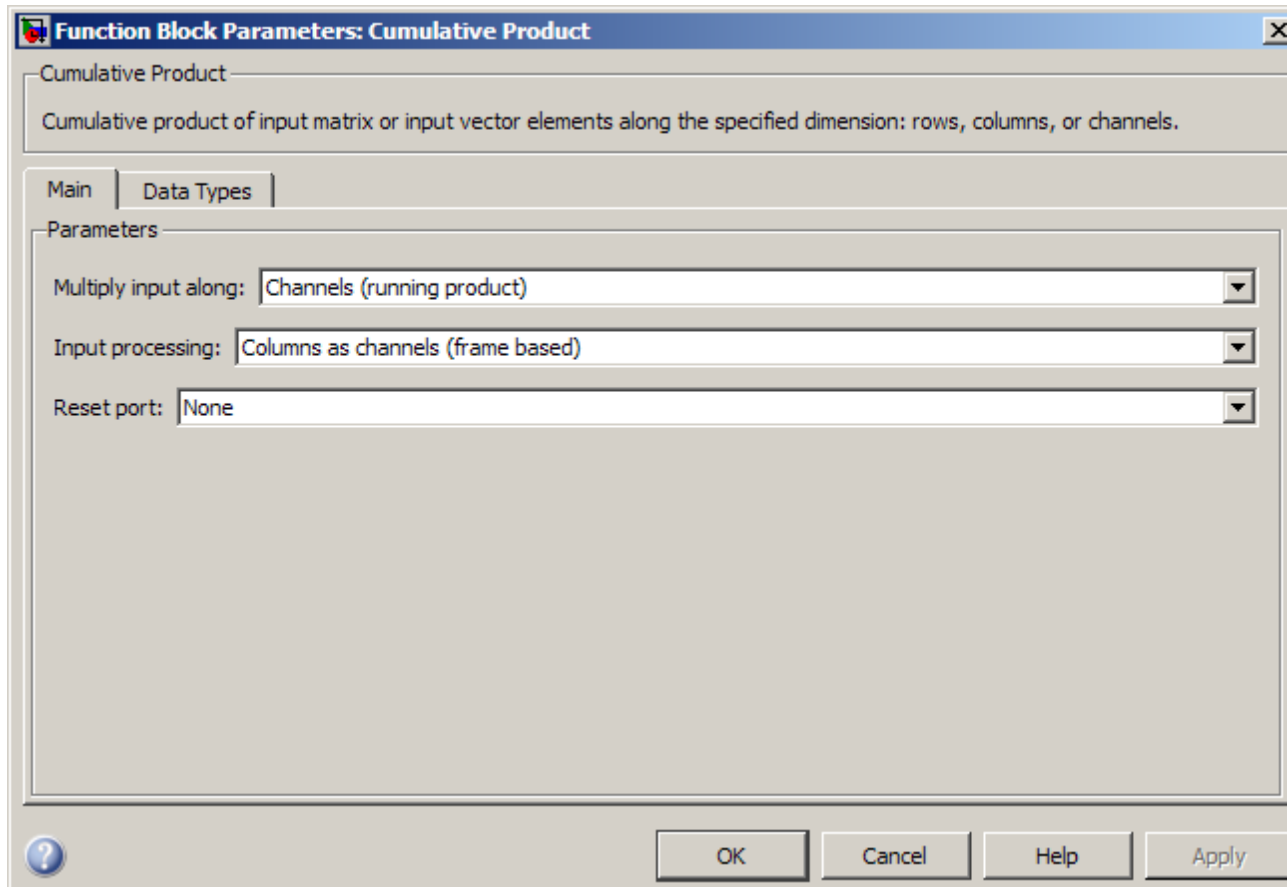
Cumulative Product

Multiply input along

Specify the dimension along which to compute the cumulative product. You can choose to multiply along `Channels (running product)`, `Columns`, or `Rows`. For more information, see the following sections:

- "Computing the Running Product Along Channels of the Input" on page 2-341

- "Multiplying Along Columns" on page 2-346

- "Multiplying Along Rows" on page 2-347

Input processing

Specify how the block should process the input when computing the running product along the channels of the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

This parameter is available only when you set the **Multiply input along** parameter to `Channels (running product)`.

Reset port

Determines the reset event that causes the block to reset the product along channels. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The

sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you set the **Multiply input along** parameter to `Channels` (`running product`). For more information, see Resetting the Running Product on page 344.

The **Data Types** pane of the Cumulative Product block dialog appears as follows.
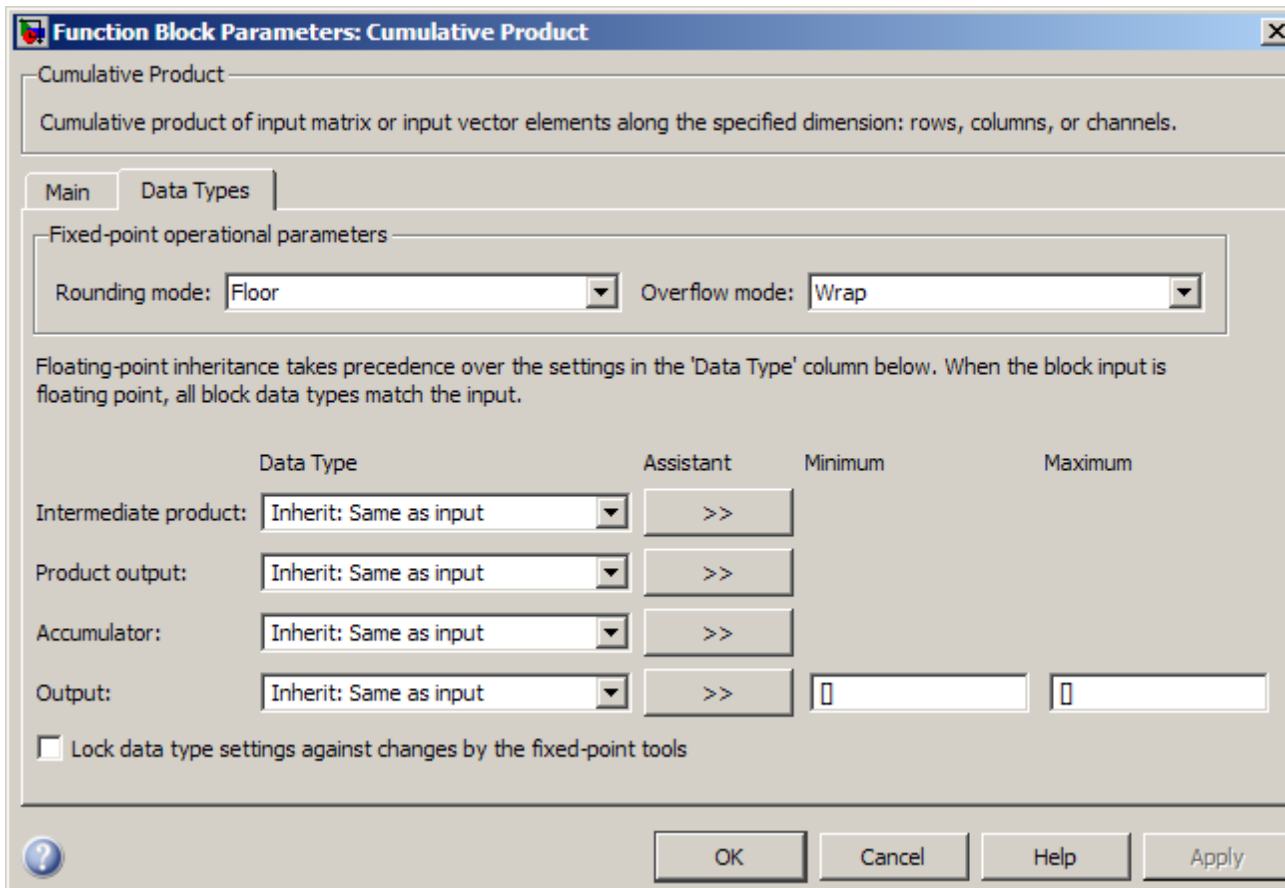
Cumulative Product

> **Note** Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode
> Select the rounding mode for fixed-point operations.
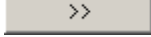
Overflow mode
> Select the overflow mode for fixed-point operations.

Intermediate product
> Specify the intermediate product data type. As shown in "Fixed-Point Data Types" on page 2-348, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. You can set it to:
>
> - A rule that inherits a data type, for example, Inherit: Same as input
>
> - An expression that evaluates to a valid data type, for example, fixdt([],16,0)
>
> Click the **Show data type assistant** button ⎡ ›› ⎤ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.
>
> See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Product output data type
> Specify the product output data type. See "Fixed-Point Data Types" on page 2-348 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:
>
> - A rule that inherits a data type, for example, Inherit: Same as input

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ▸▸ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-348 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ▸▸ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-348 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Cumulative Product

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum
Specify the minimum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Maximum
Specify the maximum value that the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Input and Output Ports	Supported Data Types
Data input port, In	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers

Input and Output Ports	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers
Reset input port, `Rst`	All built-in Simulink data types:
	• Double-precision floating point
	• Single-precision floating point
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output port	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Cumulative Sum	DSP System Toolbox
Matrix Product	DSP System Toolbox
`cumprod`	MATLAB

Cumulative Sum

Purpose   Cumulative sum of channel, column, or row elements

Library   Math Functions / Math Operations

dspmathops
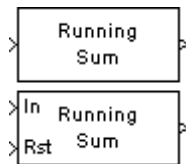
Description   The Cumulative Sum block computes the cumulative sum along the specified dimension of the input or across time (running sum).

The inputs can be a vector or a matrix. The output always has the same dimensions, rate, data type, and complexity as the input.

Input and Output Characteristics

Valid Input

The Cumulative Sum block accepts vector or matrix inputs containing real or complex values.

Valid Reset Signal

The optional reset port, Rst, accepts scalar values, which can be any built-in Simulink data type including boolean. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time.

Output Characteristics

The output always has the same dimensions, rate, data type, and complexity as the data signal input.

Computing the Running Sum Along Channels of the Input

When you set the **Sum input along** parameter to Channels (running sum), the block computes the cumulative sum of the elements in each input channel. The running sum of the current input takes into account the running sum of all previous inputs. In this mode, you must also specify a value for the **Input processing** parameter. When you set the **Input processing** parameter to Columns as channels (frame based), the block computes the running sum along each column of the current input. When you set the **Input processing** parameter to Elements as channels (sample based), the block computes

a running sum for each element of the input across time. See the
following sections for more information:

- Computing the Running Sum for Each Column of the Input on page
  357

- Computing the Running Sum for Each Element of the Input on page
  358

- Resetting the Running Sum on page 359

Computing the Running Sum for Each Column of the Input

When you set the **Input processing** parameter to `Columns as
channels (frame based)`, the block treats each input column as an
independent channel. As the following figure and equation illustrate,
the output has the following characteristics:

- The first row of the first output is the same as the first row of the
  first input.

- The first row of each subsequent output is the sum of the first row
  of the current input (time t), and the last row of the previous output
  (time t - T_f, where T_f is the frame period).

- The output has the same size, dimension, data type, and complexity
  as the input.

Given an M-by-N matrix input, u, the output, y, is an M-by-N matrix
whose first row has elements

$$y_{1,j}(t) = u_{1,j}(t) + y_{M,j}(t - T_f)$$

Cumulative Sum

Running Sum for Each Column of the Input
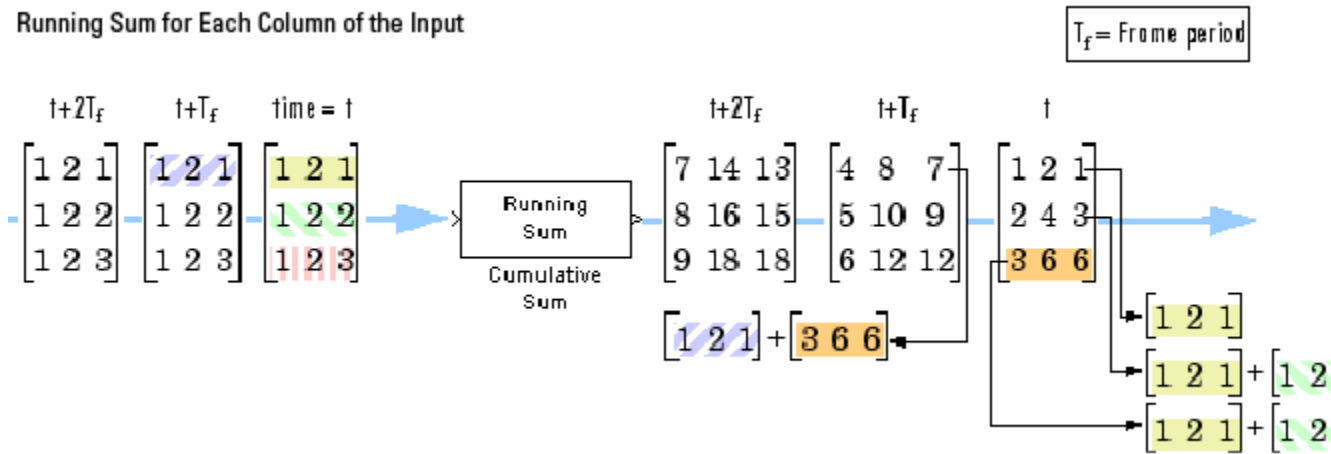
Computing the Running Sum for Each Element of the Input

When you set the **Input processing** parameter to Elements as channels (sample based), the block treats each element of the input matrix as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first output is the same as the first input.

- Each subsequent output is the sum of the current input (time t) and the previous output (time t - T_s, where T_s is the sample period).

- The output has the same size, dimension, data type, and complexity as the input.

Given an M-by-N matrix input, u, the output, y, is an M-by-N matrix with the elements

$$y_{i,j}(t) = u_{i,j}(t) + y_{i,j}(t - T_s) \quad \begin{matrix} 1 \le i \le M \\ 1 \le j \le N \end{matrix}$$

Running Sum for Each Element of the Input
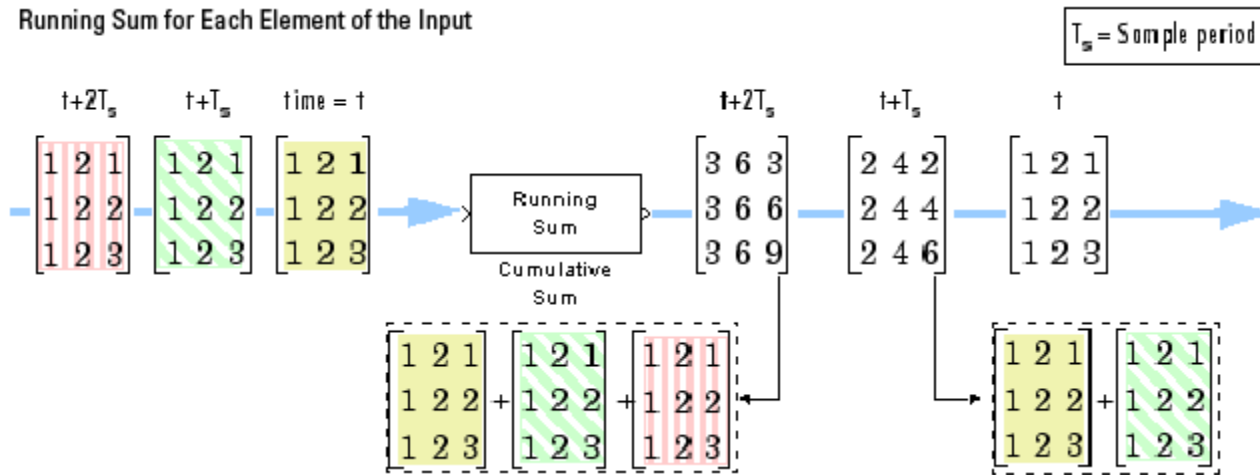
Resetting the Running Sum

When you are computing the running sum, you can configure the block to reset the running sum whenever it detects a reset event at the optional Rst port. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. The reset sample time must be a positive integer multiple of the input sample time. The input to the Rst port can be of the boolean data type.

If a reset event occurs while the block is performing sample-based processing, the block initializes the current output to the values of the current input. If a reset event occurs while the block is performing frame-based processing, the block initializes the first row of the current output to the values in the first row of the current input.

The **Reset port** parameter specifies the reset event, which can be one of the following:

• None disables the Rst port.

Cumulative Sum

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

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

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

Not a rising edge because it continues a rise from a negative value to zero.

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

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

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

Falling edge   Falling edge

Falling edge   Falling edge   **Not a falling edge** because it continues a fall from a positive value to zero.

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

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

---

Note When you run simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Thus, when the block detects a reset event, a one-sample delay occurs at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and the topic on models with multiple sample rates in the Simulink Coder documentation.

Summing Along Columns

When you set the **Sum input along** parameter to `Columns`, the block computes the cumulative sum of each column of the input. In this mode, the current cumulative sum is independent of the cumulative sums of previous inputs.

```
y = cumsum(u)     % Equivalent MATLAB code
```

The output has the same size, dimension, data type, and complexity as the input. The mth output row is the sum of the first m input rows.

Cumulative Sum

Given an M-by-N input, u, the output, y, is an M-by-N matrix whose jth column has elements

$$y_{i,j} = \sum_{k=1}^{j} u_{k,j} \qquad 1 \le i \le M$$

The block treats length-M unoriented vector inputs as M-by-1 column vectors when summing along columns.

Sum Along Columns
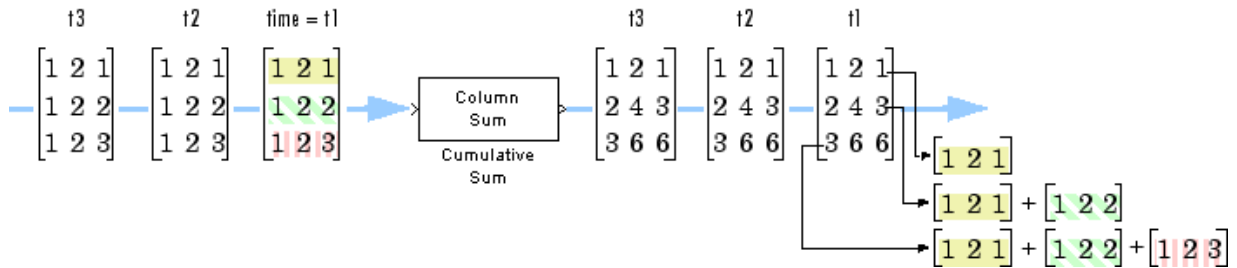
Summing Along Rows

When you set the **Sum input along** parameter to Rows, the block computes the cumulative sum of the row elements. In this mode, the current cumulative sum is independent of the cumulative sums of previous inputs.

```
y = cumsum(u,2)      % Equivalent MATLAB code
```

The output has the same size, dimension, and data type as the input. The nth output column is the sum of the first n input columns.

Given an M-by-N input, u, the output, y, is an M-by-N matrix whose ith row has elements

$$y_{i,j} = \sum_{k=1}^{j} u_{i,k} \qquad 1 \le j \le N$$

When you sum along rows, the block treats length-N unoriented vector inputs as 1-by-N row vectors.

Sum Along Rows
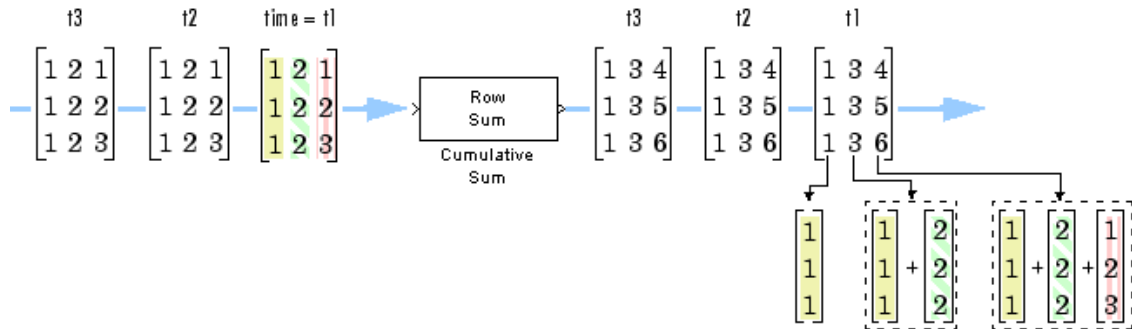
Fixed-Point Data Types

The following diagram shows the data types used within the Cumulative Sum block for fixed-point signals.
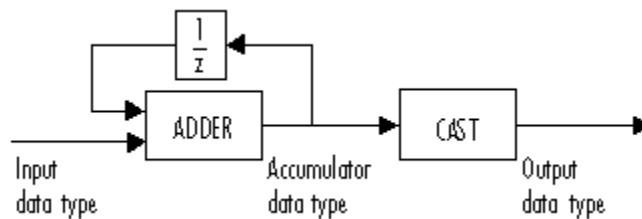
You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 2-363.

Dialog Box

The **Main** pane of the Cumulative Sum block dialog appears as follows.
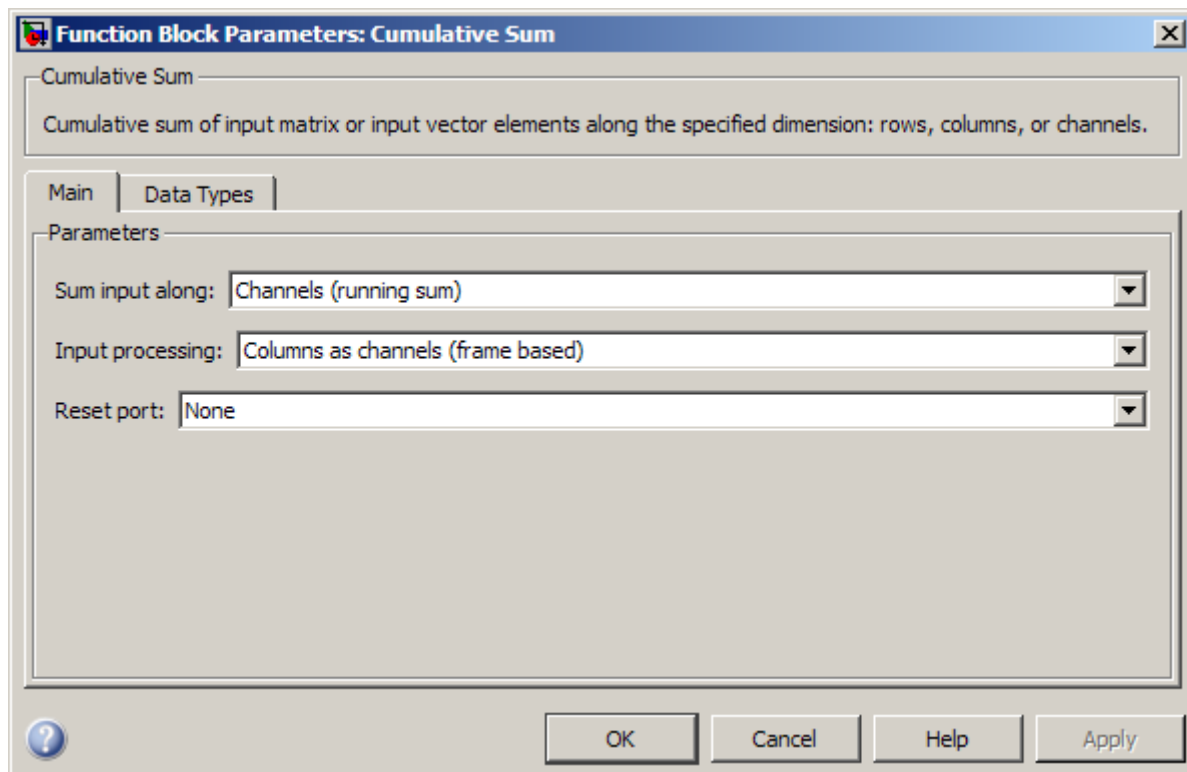
Cumulative Sum

Sum input along

Specify the dimension along which to compute the cumulative summations. You can choose to sum along Channels (running sum), Columns, or Rows. For more information, see the following sections:

- "Computing the Running Sum Along Channels of the Input" on page 2-356

- "Summing Along Columns" on page 2-361

- "Summing Along Rows" on page 2-362

Input processing

Specify how the block should process the input when computing the running sum along the channels of the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

> **Note** The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

This parameter is available only when you set the **Sum input along** parameter to `Channels (running sum)`.

Reset port

Determines the reset event that causes the block to reset the sum along channels. The rate of the input to the Rst port must be the same or slower than that of the input data signal. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you set the **Sum input along** parameter to `Channels (running sum)`. For more information, see Resetting the Running Sum on page 359.

The **Data Types** pane of the Cumulative Sum block dialog appears as follows.

Cumulative Sum

Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode
Select the rounding mode for fixed-point operations.

Overflow mode
Select the overflow mode for fixed-point operations.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-363 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit:  Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-363 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit:  Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")
- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Input and Output Ports	Supported Data Types
Data input port, `In`	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Reset input port, `Rst`	All built-in Simulink data types: • Double-precision floating point • Single-precision floating point • Boolean

Input and Output Ports	Supported Data Types
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output port	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Cumulative Product	DSP System Toolbox
Difference	DSP System Toolbox
Matrix Sum	DSP System Toolbox
cumsum	MATLAB

Data Type Conversion

Purpose   Convert input signal to specified data type

Library   Signal Management / Signal Attributes

dspsigattribs

Description   The Data Type Conversion block is an implementation of the Simulink
Data Type Conversion block. See Data Type Conversion for more
information.

Purpose   Convert magnitude data to decibels (dB or dBm)

Library   Math Functions / Math Operations

              dspmathops

Description

```
  dB
(1 ohm)
```
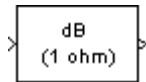
The dB Conversion block converts a linearly scaled power or amplitude input to dB or dBm. The reference power is 1 Watt for conversions to dB and 1 mWatt for conversions to dBm. The **Input signal** parameter specifies whether the input is a power signal or a voltage signal, and the **Convert to** parameter 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 output is the same size 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.

dB Conversion

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.
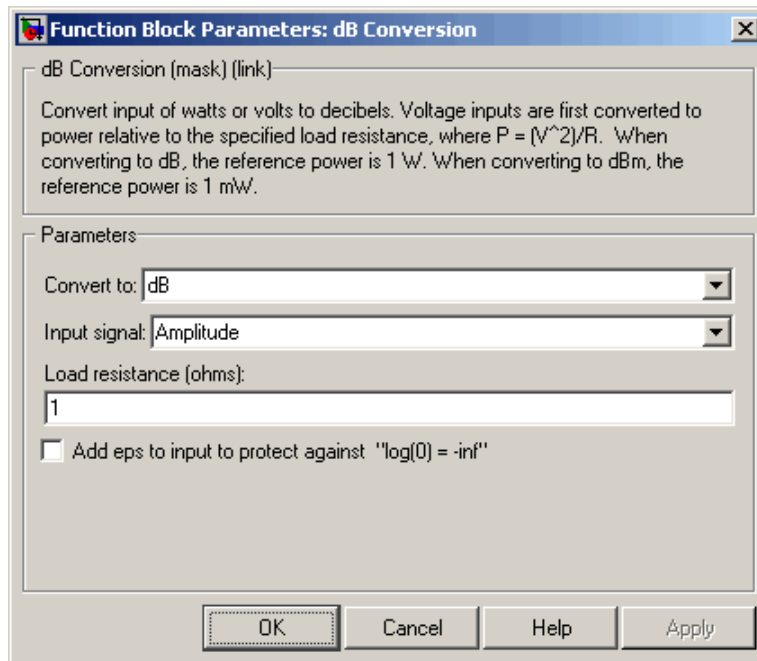
**Dialog
Box**

Convert to
>   The logarithmic scaling to which the input is converted, dB or dBm.
>   The reference power is 1 W for conversions to dB and 1 mW for
>   conversions to dBm. Tunable.

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).
>   Tunable.

dB Conversion

Supported Data Types

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

See Also

dB Gain	DSP System Toolbox
Math Function	Simulink
log10	MATLAB

Purpose

Apply decibel gain

Library

Math Functions / Math Operations

dspmathops

Description
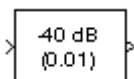
The dB Gain block multiplies the input by the decibel values specified in the **Gain** parameter. For an *M*-by-*N* input matrix *u* with elements u_{ij}, the **Gain** parameter can be a real *M*-by-*N* matrix with elements g_{ij} to be multiplied element-wise with the input, or a real scalar.

$$y_{ij} = 10 u_{ij}^{(g_{ij}/k)}$$

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 output is the same size as the input.

The dB Gain block supports real and complex floating-point and fixed-point data types.

Fixed-Point Data Types

The following diagram shows the data types used within the dB Gain subsystem block for fixed-point signals.
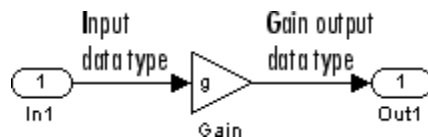
The settings for the fixed-point parameters of the Gain block in the diagram above are as follows:

dB Gain

- Integer rounding mode: `Floor`

- Saturate on integer overflow — unselected

- Parameter data type mode — `Inherit via internal rule`

- Output data type mode — `Inherit via internal rule`

See the Gain reference page for more information.
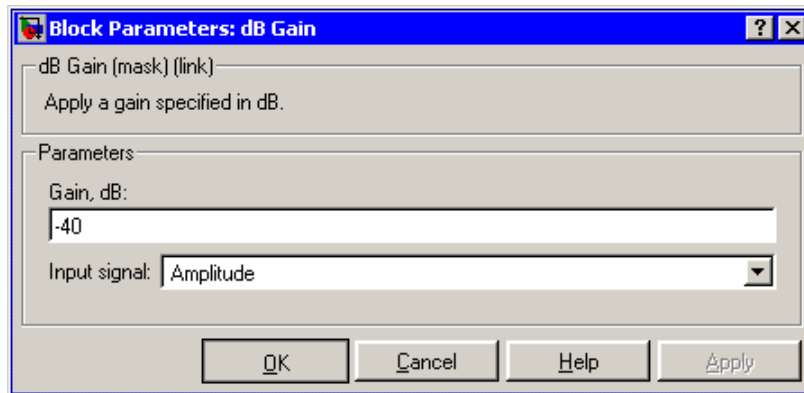
Dialog Box

Gain
:   The dB gain to apply to the input, a scalar or a real *M*-by-*N* matrix. Tunable.

Input signal
:   The type of input signal: `Power` or `Amplitude`. Tunable.

Note This block does not support tunability in generated code.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed and unsigned)

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

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

See Also

dB Conversion	DSP System Toolbox
Math Function	Simulink
log10	MATLAB

DCT

Purpose

Discrete cosine transform (DCT) of input

Library

Transforms

dspxfrm3

Description
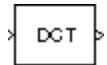
The DCT block computes the unitary discrete cosine transform (DCT) of each channel in the *M*-by-*N* input matrix, *u*.

```
y = dct(u)     % Equivalent MATLAB code
```

When the input is a sample-based row vector, the DCT block computes the discrete cosine transform across the vector dimension of the input. For all other N-D input arrays, the block computes the DCT across the first dimension. The size of the first dimension (frame size), must be a power of two. To work with other frame sizes, use the Pad block to pad or truncate the frame size to a power-of-two length.

When the input to the DCT block is an *M*-by-*N* matrix, the block treats each input column as an independent channel containing *M* consecutive samples. The block outputs an *M*-by-*N* matrix whose *l*th column contains the length-*M* DCT of the corresponding input column.

$$y(k,l) = w(k) \sum_{m=1}^{M} u(m,l) \cos \frac{\pi(2m-1)(k-1)}{2M}, \quad k = 1,...,M$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\dfrac{2}{M}}, & 2 \le k \le M \end{cases}$$

The **Sine and cosine computation** parameter determines how the block computes the necessary sine and cosine values. This parameter has two settings, each with its advantages and disadvantages, as described in the following table.

Sine and Cosine Computation Parameter Setting	Sine and Cosine Computation Method	Effect on Block Performance
Table lookup	The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block in a speed-optimized table, and retrieves the values during code execution.	The block usually runs much more quickly, but requires extra memory for storing the precomputed trigonometric values.
Trigonometric fcn	The block computes sine and cosine values during the simulation. When you generate code from the block, the processor running the generated code computes the sine and cosine values while the code runs.	The block usually runs more slowly, but does not need extra data memory. For code generation, the block requires a support library to emulate the trigonometric functions, increasing the size of the generated code.

This block supports Simulink virtual buses.

Fixed-Point Data Types

The following diagrams show the data types used within the DCT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the DCT block dialog as discussed in "Dialog Box" on page 2-382.

Inputs to the DCT block are first cast to the output data type and stored in the output buffer. Each butterfly stage processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type.

a+b

b   a-b

butterfly stage

a   a+b   W1   W1(a+b)   W1 and W2 are twiddle table constants

OR

b   a-b   W2   W2(a-b)

butterfly stage   twiddle multiplication

a   c   ac+be
                        e

OR   f

b   d   af+bd

butterfly stage

Butterfly Stage Data Types

a   a   a+b   a+b
b   CAST   b   ADDER   a-b   CAST   a-b

Inputs to   Accumulator   Accumulator   Output data
butterfly -   data type   data type   type
output
data type

Twiddle Multiplication Data Types

Input to twiddle   MULTIPLIER   CAST
multiplication -
output data type   Sine table   Accumulator or   Output
                    data type   Product output   data type
                                          data type

The output of the multiplier is in the product output data type when
at least one of the inputs to the multiplier is real. When both of the

inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types".

Note   When the block input is fixed point, all internal data types are signed fixed point.

Dialog Box

The **Main** pane of the DCT block dialog appears as follows.

Function Block Parameters: DCT

DCT

Compute the discrete cosine transform (DCT) across the first dimension of the input. The first dimension of t have a power-of-two length.

| Main | Data Types |

Parameters

Sine and cosine computation: Table lookup

OK   Cancel   Help

Sine and cosine computation
Sets the block to compute sines and cosines by either looking up sine and cosine values in a speed-optimized table (Table lookup),

or by making sine and cosine function calls (`Trigonometric fcn`).

See the table in the "Description [DCT]" on page 2-378 section.

The **Data Types** pane of the DCT block dialog appears as follows.

Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, Inherit: Same word length as input

- An expression that evaluates to a valid data type, for example, fixdt(1,16)

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; instead, they are always saturated and rounded to Nearest.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-379 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule

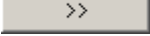- An expression that evaluates to a valid data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-379 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-379 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.

  When you select `Inherit:  Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The internal rule first calculates an ideal output word length and fraction length using the following equations:

  $$WL_{ideal\ output} = WL_{input} + floor(\log_2(DCT\ length - 1)) + 1$$

  $$FL_{ideal\ output} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see "Inherit via Internal Rule".

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button $\boxed{\quad \gg \quad}$ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

See Also

Complex Cepstrum	DSP System Toolbox
FFT	DSP System Toolbox
IDCT	DSP System Toolbox
Real Cepstrum	DSP System Toolbox
dct	Signal Processing Toolbox

Delay

Purpose   Delay discrete-time input by specified number of samples or frames

Library   Signal Operations

           `dspsigops`

Description
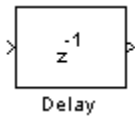
The Delay block delays a discrete-time input by the number of samples or frames specified in the **Delay units** and **Delay** parameters. The **Delay** value must be an integer value greater than or equal to zero. When you enter a value of zero for the **Delay** parameter, any initial conditions you might have entered have no effect on the output.

The Delay block allows you to set the initial conditions of the signal that is being delayed. The initial conditions must be numeric.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each column of the *M*-by-*N* input matrix as an independent channel. The block delays each channel of the input as specified by the **Delay** parameter.

The **Delay** parameter can be a scalar integer by which the block equally delays all channels or a vector whose length is equal to the number of channels.

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be constant or varying along each channel. See the "Frame-Based Processing Examples" on page 2-391 section for more information.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the N-D input array as an independent channel. Thus, the total number of channels in the input is equal to the product of the input dimensions. The dimension of the output is the same as that of the input.

The **Delay** parameter can be a scalar integer by which to equally delay all channels or an N-D array of the same dimensions as the input array,

containing nonnegative integers that specify the number of sample intervals to delay each channel of the input.

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be the same or different within a channel. See the "Sample-Based Processing Examples" on page 2-395 section for more information.

Resetting the Delay

The Delay block resets the delay whenever it detects a reset event at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

The reset event is specified by the **Reset port** parameter, and can be one of the following:

- None disables the Rst port.

- Rising edge triggers a reset operation when the Rst input does one of the following:

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

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)
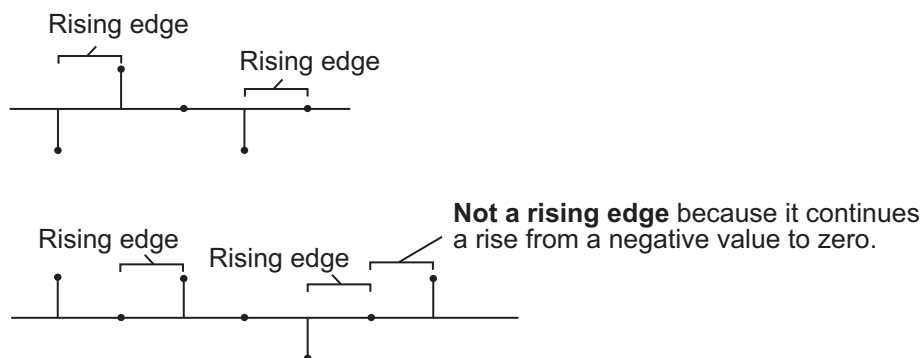
- `Falling edge` triggers a reset operation when the `Rst` input does one of the following:

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

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)
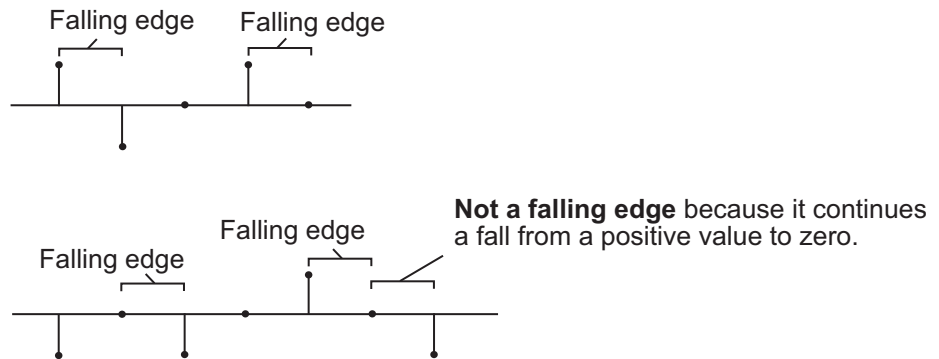
- `Either edge` triggers a reset operation when the `Rst` input is `Rising edge` or `Falling edge` (as described earlier).

- `Non-zero sample` triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink MultiTasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

This block supports Simulink virtual buses.

Examples   **Frame-Based Processing Examples**

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be constant or varying along each channel. The next sections describe the behavior of the block for each of these four cases:

Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel

Enter a scalar value for the initial conditions. This value is used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to Columns as channels (frame based).

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix},...$$

You want the initial conditions of your three-channel signal to be identical and zero for the first frame:

1 Set the **Delay (frames)** parameter to 1.

2 Clear the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.

3 Set the **Initial conditions** parameter to a scalar value of 0.

The output of the delay block is

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

0, the scalar initial condition value, is used across the channels and within the channels for the first frame. This frame is the output at sample time zero.

Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel

The initial conditions must be a vector of length N, where $N \geq 1$. N is also equal to the number of channels in your signal. These initial condition values are used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to Columns as channels (frame based).

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

You want the initial conditions of your three-channel signal to be [0 10 20] for the first frame:

1 Set the **Delay (frames)** parameter to 1.

2 Select the **Specify different initial conditions for each channel** check box.

3 Clear the **Specify different initial conditions within a channel** check box.

4 Set the **Initial conditions** parameter to [0 10 20].

The output of the delay block is

$$\begin{bmatrix} 0 & 10 & 20 \\ 0 & 10 & 20 \\ 0 & 10 & 20 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

The initial condition vector expands to create the frame that is output at sample time zero. Different initial conditions are used for each channel, but the same initial condition value is used with a channel.

Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel

In this case, the **Delay** parameter can be a scalar integer by which to equally delay all channels or a vector whose length is equal to the number of channels. All the values of this vector must be equal.

Enter the initial conditions as a vector. These values are used as the initial condition value along each of the channels to be delayed. The initial condition vector must have length equal to the value of the **Delay (frames)** parameter multiplied by the frame length. For example, if you want to delay your signal by two frames with frame length two and an initial condition value of 3, enter your initial condition vector as [3 3 3 3].

For example, suppose your input is a matrix and you set the **Input processing** parameter to Columns as channels (frame based).

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \dots$$

You want the initial conditions of your three-channel signal to be the same along each of the channels to be delayed:

1 Set the **Delay (frame)** parameter to 1.

2 Clear the **Specify different initial conditions for each channel** check box.

3 Select the **Specify different initial conditions within a channel** check box.

4 Set the **Initial conditions** parameter to [10 20 30].

The output of the delay block is

$$\begin{bmatrix} 10 & 10 & 10 \\ 20 & 20 & 20 \\ 30 & 30 & 30 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \ldots$$

The initial condition vector defines the initial condition values within each of the three channels. The same initial conditions are used for each channel, but different initial condition values are used with a channel.

Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel

Enter a cell array for your initial condition values. Or, when you have a scalar delay value, you can enter the initial conditions as a matrix.

For example, suppose your input is a matrix and you set the **Input processing** parameter to Columns as channels (frame based).

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \ldots$$

You want the initial conditions of your three-channel signal to be different for each channel and along each channel.

1 Set the **Delay (frames)** parameter to 1.

2 Select the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.

3 Set the **Initial conditions** parameter to either [10 20 30; 40 50 60; 70 80 90] or {[10 40 70];[20 50 80];[30 60 90]}. Each cell of the cell array represents the delay along one channel.

Regardless of whether you use a matrix or cell array, the output of the delay block is

$$
\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix} \dots
$$

The initial condition matrix is the output at sample time zero. The elements of the initial condition cell array define the initial condition values within each channel. The first element, a vector, represents the initial conditions within channel 1. The second element, a vector, represents the initial conditions within channel 2, and so on. Different initial conditions are used for each channel and within the channels.

Sample-Based Processing Examples

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be the same or different along each channel. The next sections describe the behavior of the block for each of these four cases:

- "Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel" on page 2-396

- "Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel" on page 2-397

- "Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel" on page 2-398

- "Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel" on page 2-399

Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel

Enter a scalar value for the initial conditions. This value is used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to Elements as channels (sample based).

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

You want the initial conditions of your four-channel signal to be identical and zero for the first two samples:

1 Set the **Delay (samples)** parameter to 2.

2 Clear the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes.

3 Set the **Initial conditions** parameter to a scalar value of 0.

The output of the delay block is

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

0, the scalar initial condition value, is used for each channel and within the channels. It is the output at sample time zero and sample time one.

Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel

The initial conditions must be an N-D array for N-D input. The initial conditions must have the same dimensions as the input data. These initial condition values are used as the constant initial condition value for each of the channels.

For example, suppose your input is a matrix and you set the **Input processing** parameter to Elements as channels (sample based).

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

You want the initial conditions of your four-channel signal to be

$$\begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}$$

for the first two samples:

1 Set the **Delay (samples)** parameter to 2.

2 Select the **Specify different initial conditions for each channel** check box.

3 Clear the **Specify different initial conditions within a channel** check box.

4 Set the **Initial conditions** parameter to [7 9; 11 13].

The output of the delay block is

$$\begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}, \begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

The initial condition matrix is the output at sample time zero and sample time one. Different initial conditions are used for each channel; the same initial condition value is used within a channel.

Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel

In this case, for N-D sample-based inputs, the initial conditions parameter must be a vector whose length is equal to the delay value, specified by the **Delay** parameter. The values in this vector are used as the initial condition values along each of the channels to be delayed.

For example, suppose your input is a matrix and you set the **Input processing** parameter to Elements as channels (sample based).

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

You want the initial conditions of your four channel signal to be the same along each of the channels to be delayed:

1 Set the **Delay (samples)** parameter to 2.

2 Clear the **Specify different initial conditions for each channel** check box.

3 Select the **Specify different initial conditions within a channel** check box.

4 Set the **Initial conditions** parameter to [10 20].

The output of the delay block is

$$\begin{bmatrix} 10 & 10 \\ 10 & 10 \end{bmatrix}, \begin{bmatrix} 20 & 20 \\ 20 & 20 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

The first element of the initial conditions vector is the output, for all channels, at sample time zero. The second element of the initial

conditions vector is the output, for all channels, at sample time one. The same initial conditions are used for each channel, but different initial condition values are used within a channel.

Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel

Enter a cell array for your initial condition values. The cell array must be the same size as your input signal. Each cell of the cell array represents the delay values for one channel, and must be a vector of size equal to the delay value. If you have a vector or scalar input and a scalar delay value, you can enter the initial conditions as a matrix.

For example, suppose your input is a matrix and you set the **Input processing** parameter to Elements as channels (sample based).

[1 1], [2 2], [3 3],...

You want the initial conditions of your two channel signal to be different for each channel and along each channel:

1 Set the **Delay (samples)** parameter to 2.

2 Select the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes.

3 Set the **Initial conditions** parameter to [10 20; 30 40].

The output of the delay block is

[10 20], [30 40], [1 1], [2 2]...

The first row of the initial conditions vector is the output at sample time zero. The second row of the initial conditions vector is the output at sample time one. Different initial conditions are used for each channel and within the channels.

In addition, suppose your input is a matrix and you set the **Input processing** parameter to `Elements as channels (sample based)`.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \cdots$$

You want the initial conditions of your two-channel signal to be different for each channel and along each channel:

1 Set the **Delay (samples)** parameter to 2.

2 Select the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.

3 Set the **Initial conditions** parameter to `{[11 15] [12 16]; [13 17] [14 18]}`. The dimensions of the cell array match the dimensions of the input. Also, each element of the cell array represents the initial conditions within one channel.

The output of the delay block is

$$\begin{bmatrix} 11 & 12 \\ 13 & 14 \end{bmatrix}, \begin{bmatrix} 15 & 16 \\ 17 & 18 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \cdots$$

Each element of the cell array represents the initial conditions within a channel. The first element, a vector, represents the initial conditions within channel 1. The second element, a vector, represents the initial conditions within channel 2, and so on. Different initial conditions are used for each channel and within the channels.

Dialog Box

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Delay

> **Note** The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Delay units
Select whether you want to delay your input by a specified number of `Samples` or `Frames`. This parameter appears only when you set the **Input processing** parameter to `Columns as channels (frame based)`.

Delay (samples) or Delay (frames)
See "Sample-Based Processing" on page 2-388 and "Frame-Based Processing" on page 2-388 for a description of what format to use for each configuration of the block dialog.

Specify different initial conditions for each channel
Select this check box when you want the initial conditions to vary across the channels. When you do not select this check box, the initial conditions are the same across the channels.

Specify different initial conditions within a channel
Select this check box when you want the initial conditions to vary within the channels. When you do not select this check box, the initial conditions are the same within the channels.

Initial conditions
Enter a scalar, vector, matrix, or cell array of initial condition values, depending on your choice for the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes. See "Sample-Based Processing" on page 2-388 and "Frame-Based Processing" on page 2-388 for a description of what format to use for each configuration of the block dialog.

Reset port
Determines the reset event that causes the block to reset the delay. For more information, see "Resetting the Delay" on page 2-389.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

Unit Delay	Simulink
Variable Fractional Delay	DSP System Toolbox
Variable Integer Delay	DSP System Toolbox

Delay Line

Purpose   Rebuffer sequence of inputs

Library   Signal Management / Buffers
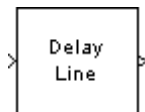
dspbuff3

Description   The Delay Line block rebuffers a sequence of M_i-by-N matrix inputs into a sequence of M_o-by-N matrix outputs, where M_o is the output frame size you specify in 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. The block always performs frame-based processing and rebuffers each of the N input channels 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 you specify in the **Initial conditions** parameter (see "Initial Conditions" on page 2-405).

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 most typical use, each output differs from the preceding output by only one sample, as illustrated below for scalar input.
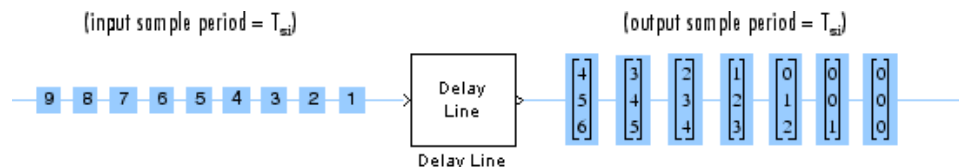
Note that the first output of the block in the example above is all zeros; this is because the **Initial Conditions** parameter is set to zero.

Initial Conditions

The Delay Line block's buffer is initialized to the value specified by the **Initial conditions** parameter. The block outputs this buffer at the first simulation step (*t*=0). When the block's output is a vector, the **Initial conditions** can be a vector of the same size, or a scalar value to be repeated across all elements of the initial output. When the block's output is a matrix, the **Initial conditions** can be a matrix of the same size, a vector (of length equal to the number of matrix rows) to be repeated across all columns of the initial output, or a scalar to be repeated across all elements of the initial output.

Examples   In the following ex_delayline_ref2 model, the block rebuffers a two-channel input with a **Delay line size** of 3.
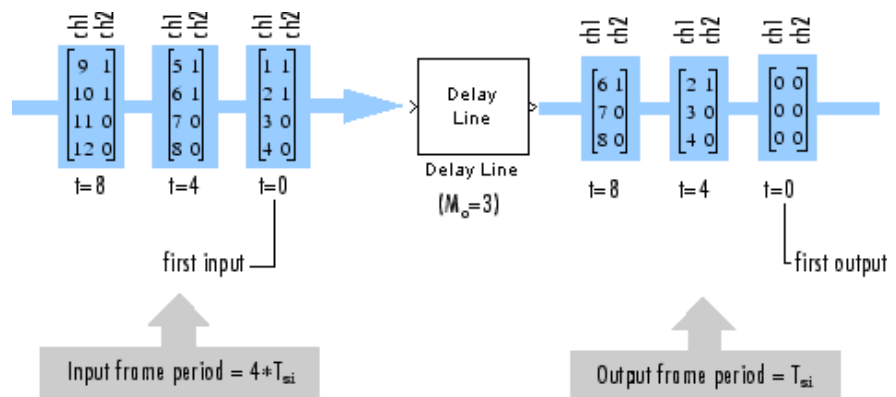
The first output frame in this example is due to the latency of the Delay Line block; it is all zeros because the **Initial conditions** parameter is set to zero. Because the input frame size of 4 is larger than the output frame size of 3, only the last three samples in each input frame are propagated 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.

Delay Line
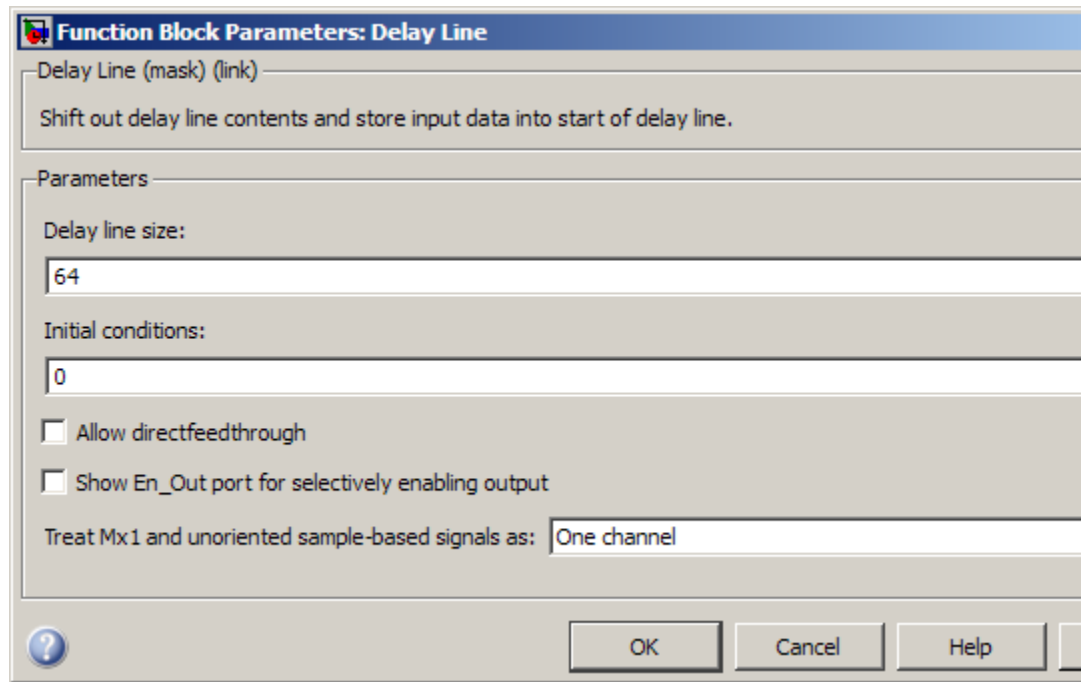
Dialog Box

Delay line size
  Specify the number of rows in output matrix, M_o.

Initial conditions
  Specify the value of the block's initial output. When the block outputs a vector, the **Initial conditions** can be a vector of the same size, or a scalar value to be repeated across all elements of the initial output. When the block outputs a matrix, the **Initial conditions** 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.

Allow direct feedthrough

When you select this check box, the input data is not delayed by an extra frame before it is available at the output buffer. Instead, the input data is available immediately at the output port of the block.

Show En_Out port for selectively enabling output

When you select this check box, the En_Out port appears on the block icon. This block uses a circular buffer internally even though the output is linear. This means that for valid output, data from the circular buffer has to be linearized. The En_Out port determines whether or not a valid output needs to be computed based on the value of its `Boolean` input. If the input value to the En_Out port is `1`, the block output is linearized, and thus is valid. Otherwise, the output is not linearized, and is invalid. This allows the block to be more efficient when the tapped Delay Line's output is not required at each sample time.

Note that when the input value to the En_Out port is `0`, the block can give different results depending on the state of the model. The results can appear to match valid results or can be invalid, and they cannot be predicted. You should ignore the block output in all cases when the input to the En_Out port is `0`.

Hold previous value when the output is disabled

This parameter only appears and applies when the **Show En_Out port for selectively enabling output** parameter is selected. Use this parameter to specify the block output at those time steps when the internal state buffer is not being linearized to output valid data.

When you do not select this check box, the block memory is free to be used by other parts of the model, and the signal on the output port is invalid. When you select this check box, the most recent valid value is held on the output port, and slightly more memory is used by the block.

Treat Mx1 and unoriented sample-based signals as

Specify how the block treats sample-based *M*-by-1 column vectors and unoriented sample-based vectors of length *M*. You can select one of the following options:

- One channel — When you select this option, the block treats *M*-by-1 and unoriented sample-based inputs as a column vector (one channel).

- M channels (this choice will be removed — see release notes) — When you select this option, the block treats *M*-by-1 and unoriented sample-based inputs as a 1-by-*M* row vector. Because the block always does frame-based processing, the block interprets the 1-by-*M* row vector as *M* individual channels.

Note This parameter will be removed in a future release. At that time, the Delay Line block will always perform frame-based processing.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned)

Port	Supported Data Types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Buffer   DSP System Toolbox

Triggered Delay Line   DSP System Toolbox
(Obsolete)

Detrend

Purpose   Remove linear trend from vectors

Library   Statistics

dspstat3
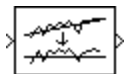
Description   The Detrend block removes a linear trend from the length-*M* input
vector, *u*, by subtracting the straight line that best fits the data in the
least squares sense.

The least squares line, $\hat{u} = ax + b$, is the line with parameters *a* and *b*
that minimizes the quantity

$$\sum_{i=1}^{M} (u_i - u_i)^2$$

for *M* evenly-spaced values of *x*, where u_i is the *i*th element in the input
vector. The output, $y = u\text{-}\hat{u}$, is always an *M*-by-1 column vector.
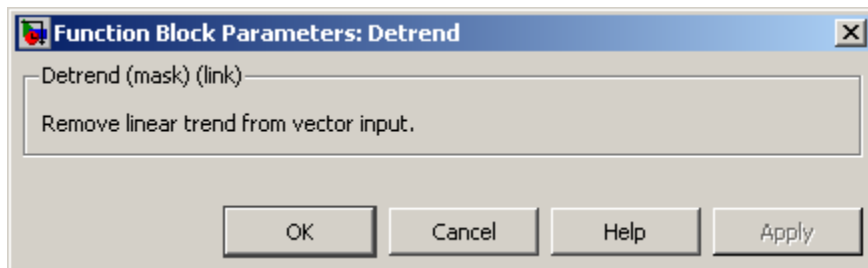
Function Block Parameters: Detrend	✕
Detrend (mask) (link)	
Remove linear trend from vector input.	
OK Cancel Help Apply	

**Dialog
Box**

**Supported
Data
Types**
- Double-precision floating point
- Single-precision floating point

See Also

Cumulative Sum	DSP System Toolbox
Difference	DSP System Toolbox
Least Squares Polynomial Fit	DSP System Toolbox
Unwrap	DSP System Toolbox
detrend	MATLAB

Difference

Purpose   Compute element-to-element difference along specified dimension of input

Library   Math Functions / Math Operations

dspmathops

Description   The Difference block computes the difference between adjacent elements in rows, columns, or a specified dimension of the input array u. You can configure the block to compute the difference only within the current input, or across consecutive inputs (running difference).

Basic Operation

When you set the **Running difference** parameter to No, the block computes the difference between adjacent elements in the specified dimension of the current input. In this mode, the block can compute the difference along the columns, rows, or a specified dimension of the input.

Columnwise Differencing

When you set the **Difference along** parameter to Columns, the block computes differences between adjacent elements in each column of the input.

```
y = diff(u)      % Equivalent MATLAB code
```

For *M*-by-*N* inputs, the output is an (*M*-1)-by-*N* matrix whose *j*th column has the following elements:

$$y_{i,j} = u_{i+1,j} - u_{i,j} \quad 1 \le i \le (M-1)$$

Rowwise Differencing

When you set the **Difference along** parameter to Rows, the block computes differences between adjacent elements in each row of the input.

```
y = diff(u,[],2)      % Equivalent MATLAB code
```

The output is an M-by-(N-1) matrix whose ith row has the following elements:

$$y_{i,j} = u_{i,j+1} - u_{i,j} \quad 1 \le j \le (N-1)$$

Differencing Along Arbitrary Dimensions

When you set the **Difference along** parameter to `Specified dimension`, the behavior of the block is an extension of the rowwise differencing described earlier. The block computes differences between adjacent elements along the dimension you specify in the **Dimension** parameter.

```
y = diff(u,[],d) % Equivalent MATLAB code where d is the dimension
```

The output is an array whose length in the specified dimension is one less than that of the input, and whose lengths in other dimensions are unchanged. For example, consider an M-by-N-by-P-by-R input array with elements $u(i,j,k,l)$ and assume that **Dimension** is `3`. The output of the block is an M-by-N-by-(P–1)-by-R array with the following elements:

$$y_{i,j,k,l} = u_{i,j,k+1,l} - u_{i,j,k,l} \quad 1 \le k \le (P-1)$$

Running Operation

When you set the **Running difference** parameter to `Yes`, the block computes the running difference along the columns of the input.

For an M-by-N input matrix, the output is an M-by-N matrix whose jth column has the following elements:

$$y_{i,j} = u_{i+1,j} - u_{i,j} \quad 2 \le i \le (M-1)$$

In running mode, the first element of the output for each column is the first input element minus the last input element of the previous frame. For the first frame, the block subtracts zero from the first input element.

$$y_{1,j}(t) = u_{1,j}(t) - u_{M,j}(t - T_f)$$

Difference

Fixed-Point Data Types

The following diagram shows the data types used within the Difference block for fixed-point signals.
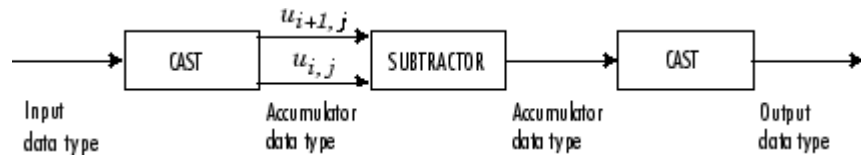
You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 2-415 .

Examples   For an example showing the modes of the Difference block, open ex_difference.

The following figure shows the output of the block when it is differencing along columns in nonrunning mode.
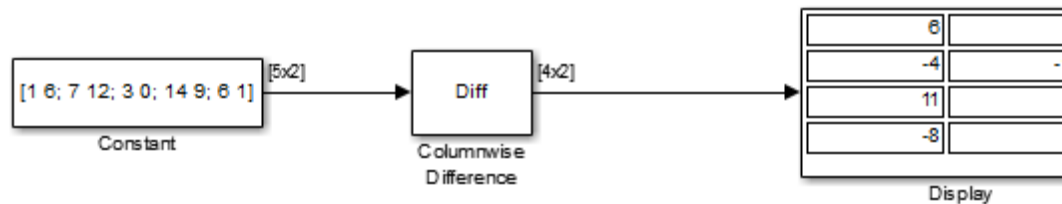
The following figure shows the output of the block when it is differencing along rows in nonrunning mode.
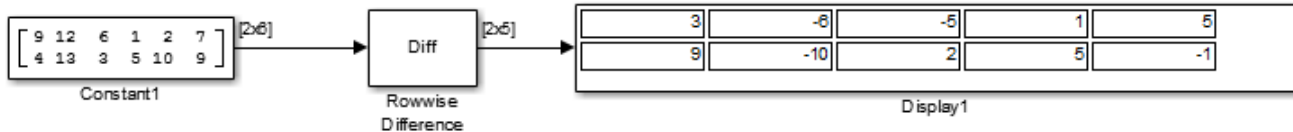
The following figure shows the second frame of block output when the block is computing the running difference.
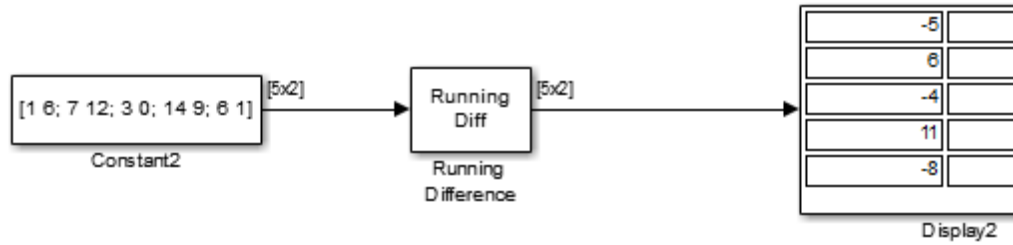
Dialog Box

The **Main** pane of the Difference block appears as follows.
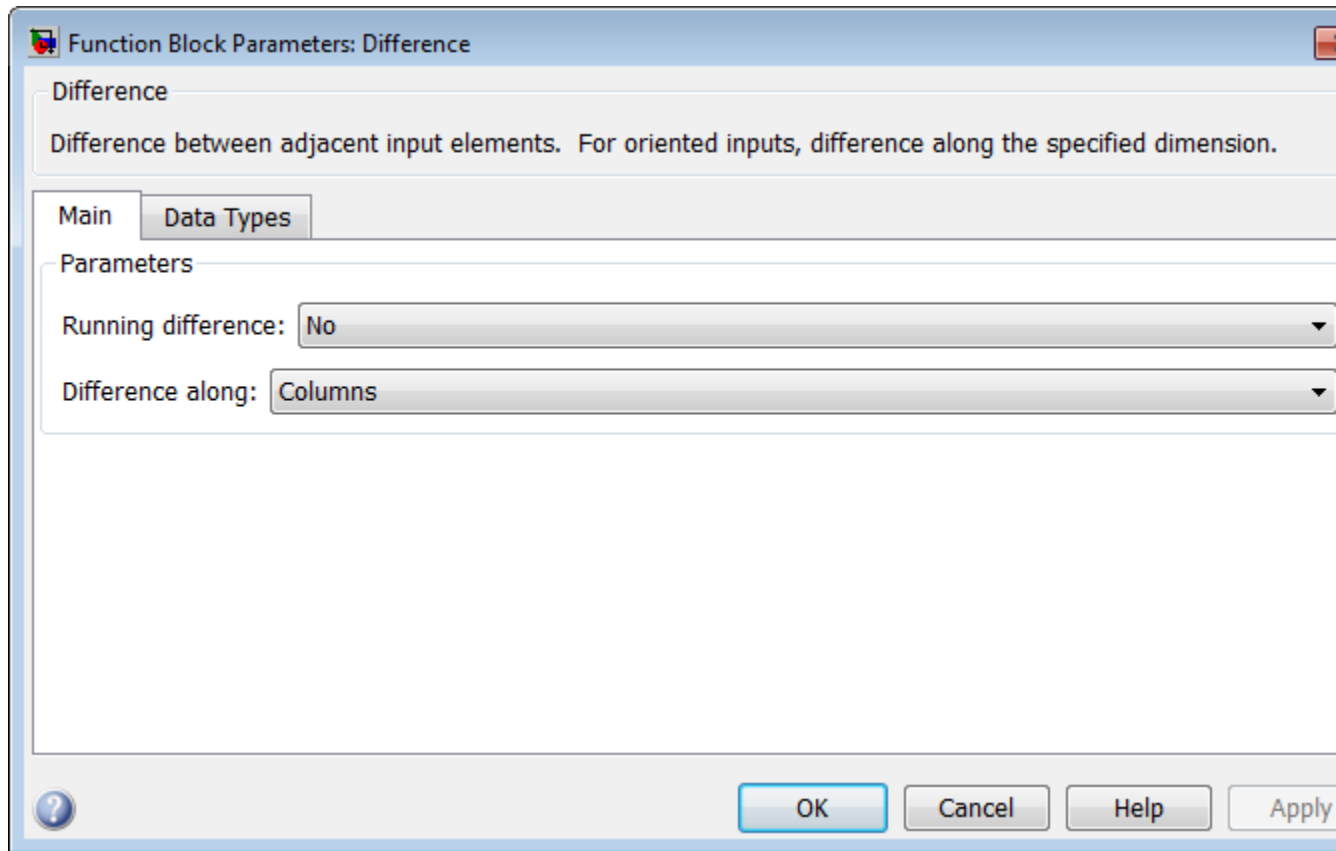
Difference

Running difference

Specify whether or not the block computes a running difference. When you select No, the block computes the difference between adjacent elements in the current input. When you select Yes, the block computes the running difference across consecutive inputs. In running mode, the block always computes the difference along the columns of the input. See "Running Operation" on page 2-413 for more information.

Note The Inherit from input (this choice will be removed - see release notes) option will be removed in a future release. See "Difference Block Changes" in the *DSP System Toolbox Release Notes* for more information.

Difference along

Specify whether the block computes the difference along the columns, rows, or specified dimension of the input.

Dimension

Specify the one-based dimension along which to compute element-to-element differences.

This parameter is only visible when you select Specified dimension for the **Difference along** parameter.

The **Data Types** pane of the Difference block appears as follows.

Difference

Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode
  Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-414 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-414 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Difference

Minimum

Specify the minimum value that the block should output. The
default value is [] (unspecified). Simulink software uses this
value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The
default value is [] (unspecified). Simulink software uses this
value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from
overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also   Cumulative Sum   DSP System Toolbox

                     diff   MATLAB

Differentiator Filter

Purpose   Design differentiator filter
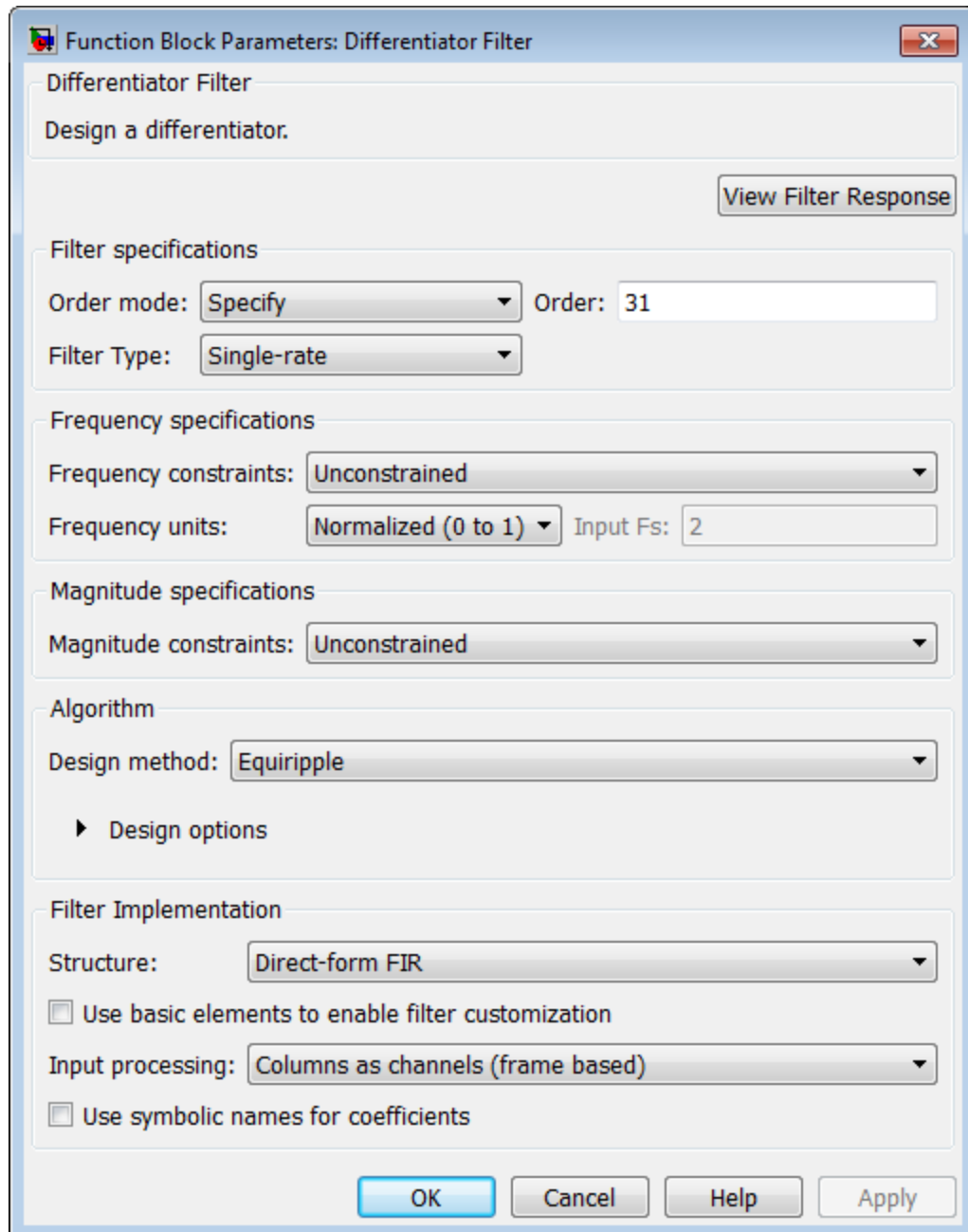
Library   Filtering / Filter Designs

dspfdesign

Description   This block brings the filter design capabilities of the `filterbuilder`
function to the Simulink environment.
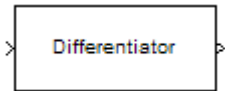
>| Differentiator |>

**Dialog
Box**   See "Differentiator Filter Design Dialog Box — Main Pane" on page
7-707 for more information about the parameters of this block. The
Data Types and **Code Generation** panes are not available for blocks
in the DSP System Toolbox Filter Designs library.

Function Block Parameters: Differentiator Filter

Differentiator Filter

Design a differentiator.

View Filter Response

Filter specifications

Order mode: Specify   Order: 31

Filter Type: Single-rate

Frequency specifications

Frequency constraints: Unconstrained

Frequency units: Normalized (0 to 1)   Input Fs: 2

Magnitude specifications

Magnitude constraints: Unconstrained

Algorithm

Design method: Equiripple

▶ Design options

Filter Implementation

Structure: Direct-form FIR

☐ Use basic elements to enable filter customization

Input processing: Columns as channels (frame based)

☐ Use symbolic names for coefficients

OK   Cancel   Help   Apply

Differentiator Filter

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

Order mode

Select either `Minimum` or `Specify` (the default). Selecting `Specify` enables the **Order** option so you can enter the filter order.

Order

Enter the filter order. This option is enabled only if you set the **Order mode** to `Specify`. The default order is 31.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

> Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Frequency constraints

> This option is only available when you specify the order of the filter design. Supported options are `Unconstrained` and `Passband edge and stopband edge`.

Frequency units

> Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

> Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

> Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop

> Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands. These parameters are only available for minimum-order designs.

Magnitude constraints
This option is only available when you specify the order of your filter design. The available **Magnitude constraints** depend on the value of the **Frequency constraints** parameter. When you set the **Frequency constraints** parameter to Unconstrained, the **Magnitude constraints** parameter must also be Unconstrained. When you set the **Frequency constraints** parameter to Passband edge and stopband edge, the **Magnitude constraints** parameter can be Unconstrained, Passband ripple, or Stopband attenuation.

Magnitude units
Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Apass
Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2
Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Wpass

Passband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to `Passband edge and stopband edge` and the **Design method** is `Equiripple`.

Wstop

Stopband weight. This option is only available for a specified-order design when **Frequency constraints** is

Differentiator Filter

equal to `Passband edge` and `stopband edge` and the
Design method is `Equiripple`.

Filter Implementation

Structure
> For the filter specifications and design method you select, this
> parameter lists the filter structures available to implement your
> filter. By default, FIR filters use direct-form structure.

Use basic elements to enable filter customization
> Select this check box to implement the filter as a subsystem of
> basic Simulink blocks. Clear the check box to implement the filter
> as a high-level subsystem. By default, this check box is cleared.
>
> The high-level implementation provides better compatibility
> across various filter structures, especially filters that would
> contain algebraic loops when constructed using basic elements.
> On the other hand, using basic elements enables the following
> optimization parameters:
>
> - **Optimize for zero gains** — Terminate chains that contain
>   Gain blocks with a gain of zero.
>
> - **Optimize for unit gains** — Remove Gain blocks that scale
>   by a factor of one.
>
> - **Optimize for delay chains** — Substitute delay chains made
>   up of n unit delays with a single delay by n.
>
> - **Optimize for negative gains** — Use subtraction in Sum
>   blocks instead of negative gains in Gain blocks.

Input processing
> Specify how the block should process the input. The available
> options may vary depending on he settings of the **Filter
> Structure** and **Use basic elements for filter customization**
> parameters. You can set this parameter to one of the following
> options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed  see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.

- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Differentiator Filter

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

Purpose   Filter each channel of input over time using static or time-varying digital filter implementations

Library   Filtering / Filter Implementations

dsparch4
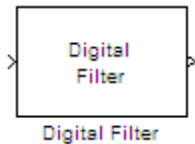
Description   You can use the Digital Filter block to efficiently implement a floating-point or fixed-point filter for which you know the coefficients, or that is already defined in a dfilt object. The block independently filters each channel of the input signal with a specified digital IIR or FIR filter. The block can implement *static filters* with fixed coefficients, as well as *time-varying filters* with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. You must set the **Input processing** parameter to specify how the block interprets the input signal. You can select one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as an independent channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as an individual channel.

The output dimensions always match those of the input signal. The outputs of this block numerically match the outputs of the Digital Filter Design block and of the dfilt object.

Note The Digital Filter block has direct feedthrough, so if you connect the output of this block back to its input you get an algebraic loop. For more information on direct feedthrough and algebraic loops, see "Algebraic Loops" in the Simulink documentation.

Digital Filter

Sections of This Reference Page

Coefficient Source

The Digital Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** Enter information about the filter such as structure and coefficients in the block mask.

- **Input port(s)** Enter the filter structure in the block mask, and the filter coefficients come in through one or more block ports. This mode is useful for specifying time-varying filters.

- **Discrete-time filter object (DFILT)** Specify the filter using a `dfilt` object.

Supported Filter Structures

When you select **Discrete-time filter object (DFILT)**, the following `dfilt` structures are supported:

- `dfilt.df1`
- `dfilt.df1t`
- `dfilt.df2`

- `dfilt.df2t`

- `dfilt.df1sos`

- `dfilt.df1tsos`

- `dfilt.df2sos`

- `dfilt.df2tsos`

- `dfilt.dffir`

- `dfilt.dffirt`

- `dfilt.dfsymfir`

- `dfilt.dfasymfir`

- `dfilt.latticear`

- `dfilt.latticemamin`

When you select **Dialog parameters** or **Input port(s)**, the list of filter structures offered in the **Filter structure** parameter depends on whether you set the **Transfer function type** to IIR (poles & zeros), IIR (all poles), or FIR (all zeros), as summarized in the following table.

Note Each structure listed in the table below supports both fixed-point and floating-point signals.

The table also shows the vector or matrix of filter coefficients you must provide for each filter structure. For more information on how to specify filter coefficients for various filter structures, see "Specify Static Filters" and "Specify Time-Varying Filters".

Digital Filter

Filter Structures and Filter Coefficients

Transfer Function Type	Supported Filter Structures	Filter Coefficient Specification

IIR (poles & zeros)	`Direct form I`
	
	`Direct form I transposed`
	
	`Direct form II`
	
	`Direct form II transposed`
		• Numerator coefficients vector [b0, b1, b2, ..., bn]
		
		• Denominator coefficients vector [a0, a1, a2, ..., am]
		
		See Special Consideration for the Leading Denominator Coefficient.

	`Biquadratic direct form I (SOS)`
	
	`Biquadratic direct form I transposed (SOS)`
	
	`Biquadratic direct form II (SOS)`
	
	`Biquadratic direct form II transposed (SOS)`
		• M-by-6 second-order section (SOS) matrix.
		
		• Scale values
		
		See "Specify the SOS Matrix (Biquadratic Filter Coefficients)".

IIR (all poles)	`Direct form`
	
	`Direct form transposed`
		Denominator coefficients vector [a0, a1, a2, ..., am]
		
		See Special Consideration for the Leading Denominator Coefficient.

	`Lattice AR`	Reflection coefficients vector [k1, k2, ..., kn]

Filter Structures and Filter Coefficients (Continued)

Transfer Function Type	Supported Filter Structures	Filter Coefficient Specification
FIR (all zeros)	Direct form Direct form symmetric Direct form antisymmetric Direct form transposed	Numerator coefficients vector [b0, b1, b2, ..., bn]
	Lattice MA	Reflection coefficients vector [k1, k2, ..., kn]

Special Considerations for the Leading Denominator Coefficient

In some cases, the Digital Filter block requires the leading denominator coefficient (a_0) to be 1. This requirement applies under the following conditions:

- The Digital Filter block is operating in a fixed-point mode. The block operates in a fixed-point mode when at least one of the following statements is true:

    - The input to the Digital Filter block has a fixed-point or integer data type.

    - The **Fixed-point instrumentation mode** parameter under **Analysis > Fixed Point Tool** has a setting of Minimums, maximums and overflows.

- The **Coefficient source** has a setting of Dialog or Input port(s).

Digital Filter

- The **Transfer function type** and **Filter structure** parameters are set to one of the combinations described in the following table.

Transfer function type	Filter structure
IIR (poles & zeros)	Direct form I
	Direct form I transposed
	Direct form II
	Direct form II transposed
IIR (all poles)	Direct form
	Direct form transposed

The Digital Filter block produces an error if you use it in one of the these configurations and your leading denominator coefficient (a_0) does not equal 1. To resolve the error, set your leading denominator coefficient to 1 by scaling all numerator and denominator coefficients by a factor of a_0.

Specifying Initial Conditions

In **Dialog parameters** and **Input port(s)** modes, the block initializes the internal filter states to zero by default, which is equivalent to assuming past inputs and outputs are zero. You can optionally use the **Initial conditions** parameter to specify nonzero initial conditions for the filter delays.

To determine the number of initial condition values you must specify, and how to specify them, see the following table on Valid Initial Conditions and Number of Delay Elements (Filter States) on page

2-439. The **Initial conditions** parameter can take one of four forms as described in the following table.

Valid Initial Conditions

Initial Condition	Examples	Description
Scalar	5 Each delay element for each channel is set to 5.	The block initializes all delay elements in the filter to the scalar value.
Vector (for applying the same delay elements to each channel)	For a filter with two delay elements: $[d_1\ d_2]$ The delay elements for all channels are d1 and d2.	Each vector element specifies a unique initial condition for a corresponding delay element. The block applies the same vector of initial conditions to each channel of the input signal. The vector length must equal the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States) on page 2-439).
Vector or matrix (for applying different delay elements to each channel)	For a 3-channel input signal and a filter with two delay elements: $[d_1\ d_2\ D_1\ D_2\ d_1\ d_2]$ or $$\begin{bmatrix} d_1 & D_1 & d_1 \\ d_2 & D_2 & d_2 \end{bmatrix}$$ • The delay elements for channel 1 are d_1 and d_2. • The delay elements for channel 2 are D_1 and D_2. • The delay elements for channel 3 are d_1 and d_2.	Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel: • The vector length must be equal to the product of the number of input channels and the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States) on page 2-439). • The matrix must have the same number of rows as the number of delay elements in the filter (specified in the table Number of Delay

Valid Initial Conditions (Continued)

Initial Condition	Examples	Description
		Elements (Filter States) on page 2-439), and must have one column for each channel of the input signal.
Empty matrix	[ ] Each delay element for each channel is set to 0.	The empty matrix, [ ], is equivalent to setting the **Initial conditions** parameter to the scalar value 0.

The number of delay elements (filter states) per input channel depends on the filter structure, as indicated in the following table.

Number of Delay Elements (Filter States)

Filter Structure	Number of Delay Elements per Channel
Direct form Direct form transposed Direct form symmetric Direct form antisymmetric	#_of_filter_coeffs-1
Direct form I Direct form I transposed	• #_of_zeros-1 • #_of_poles-1
Direct form II Direct form II transposed	max(#_of_zeros, #_of_poles)-1
Biquadratic direct form I (SOS) Biquadratic direct form I transposed (SOS) Biquadratic direct form II (SOS) Biquadratic direct form II transposed (SOS)	2 * #_of_filter_sections
Lattice AR Lattice MA	#_of_reflection_coeffs

Digital Filter

State Logging

Simulink enables you to log the states in your model to the MATLAB workspace. The following table indicates which filter structures of the Digital Filter block support the Simulink state logging feature. See "States" in the Simulink User's Guide documentation for more information.

Transfer Function Type	Filter Structure	State Logging Supported
IIR (poles & zeros)	`Direct form I`	No
	`Direct form I transposed`	Yes
	`Direct form II`	No
	`Direct form II transposed`	Yes
	`Biquadratic direct form I (SOS)`	Yes
	`Biquadratic direct form I transposed (SOS)`	Yes
	`Biquadratic direct form II (SOS)`	Yes
	`Biquadratic direct form II transposed (SOS)`	Yes
IIR (all poles)	`Direct form`	No
	`Direct form transposed`	Yes
	`Lattice AR`	Yes
FIR (all zeros)	`Direct form`	No
	`Direct form symmetric`	No
	`Direct form antisymmetric`	No
	`Direct form transposed`	Yes
	`Lattice MA`	Yes

Fixed-Point Data Types

All structures supported by the Digital Filter block support fixed-point data types. You can specify intermediate fixed-point data types for quantities such as the coefficients, accumulator, and product output for each filter structure. See "Filter Structure Diagrams" on page 2-459 for diagrams depicting the use of these intermediate fixed-point data types in each filter structure.

Dialog Box

Coefficient Source

The Digital Filter block can operate in three different modes. Select the mode in the **Coefficient source** group box.

- **Dialog parameters** Enter information about the filter such as structure and coefficients in the block mask.

- **Input port(s)** Enter the filter structure in the block mask, and the filter coefficients come in through one or more block ports. This mode is useful for specifying time-varying filters.

- **Discrete-time filter object (DFILT)** Specify the filter using a dfilt object.

Different items appear on the Digital Filter block dialog depending on whether you select **Dialog parameters**, **Input port(s)**, or **Discrete-time filter object (DFILT)** in the **Coefficient source** group box. See the following sections for details:

- "Specify Filter Characteristics in Dialog and/or Through Input Ports" on page 2-441

- "Specify Discrete-Time Filter Object" on page 2-454

Specify Filter Characteristics in Dialog and/or Through Input Ports

The **Main** pane of the Digital Filter block dialog appears as follows when **Dialog parameters** is specified in the **Coefficient source** group

Digital Filter

box. The parameters below can appear when **Dialog parameters** or **Input port(s)** is selected, as noted.

Digital Filter
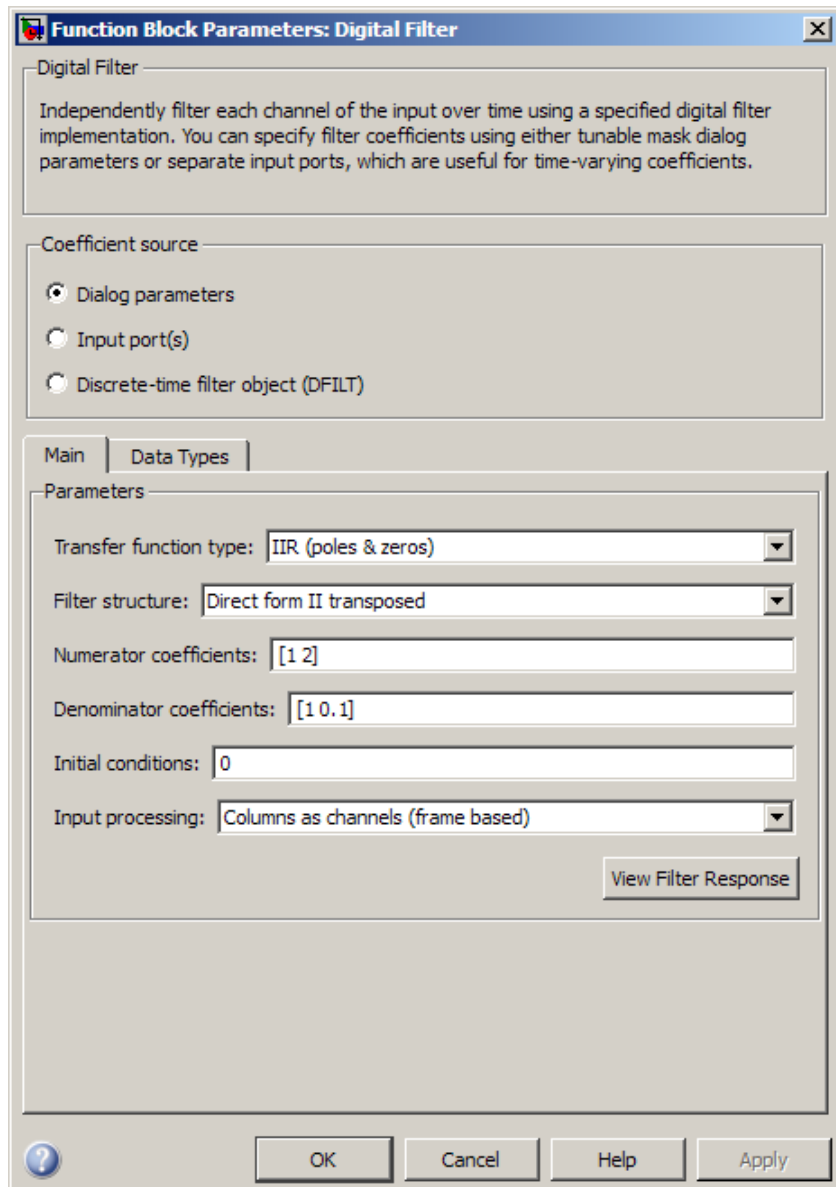
Transfer function type

Select the type of transfer function of the filter; `IIR (poles & zeros)`, `IIR (all poles)`, or `FIR (all zeros)`. See "Supported Filter Structures" on page 2-432 for more information.

Filter structure

Select the filter structure. The selection of available structures varies depending the setting of the **Transfer function type** parameter. See "Supported Filter Structures" on page 2-432 for more information.

Numerator coefficients

Specify the vector of numerator coefficients of the filter's transfer function.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure lends itself to specification with numerator coefficients. Tunable.

Denominator coefficients

Specify the vector of denominator coefficients of the filter's transfer function.

In some cases, the leading denominator coefficient (a0) must be 1. See Special Consideration for the Leading Denominator Coefficient for more information.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure lends itself to specification with denominator coefficients. Tunable.

Reflection coefficients

Specify the vector of reflection coefficients of the filter's transfer function.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure lends itself to specification with reflection coefficients. Tunable.

SOS matrix (Mx6)

Specify an M-by-6 *SOS matrix* containing coefficients of a second-order section (SOS) filter, where M is the number of sections. You can use the `ss2sos` and `tf2sos` functions from Signal Processing Toolbox software to check whether your SOS matrix is valid. For more on the requirements of the SOS matrix, see "Specify the SOS Matrix (Biquadratic Filter Coefficients)".

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure is biquadratic. Tunable.

Scale values

Specify the scale values to be applied before and after each section of a biquadratic filter.

- If you specify a scalar, that value is applied before the first filter section. The rest of the scale values are set to 1.

- You can also specify a vector with $M + 1$ elements, assigning a different value to each scale. See "Filter Structure Diagrams" on page 2-459 for diagrams depicting the use of scale values in biquadratic filter structures.

This parameter is only visible when **Dialog parameters** is selected *and* when the selected filter structure is biquadratic. Tunable.

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

Select this parameter to reduce the number of computations the block must make to produce the output by omitting the $1 / a_0$ term in the filter structure. The block output is invalid if you select this parameter when the first denominator filter coefficient is *not* always 1 for your time-varying filter.

This parameter is only enabled when the **Input port(s)** is selected *and* when the selected filter structure lends itself to this

specification. See "Removing the a0 Term in the Filter Structure" for a diagram and details.

Coefficient update rate

Specify how often the block updates time-varying filters; once per sample or once per frame.

This parameter appears only when the following conditions are met:

- You specify **Input port(s)** in the Coefficient source group box.

- You set the **Input processing** parameter to `Columns as channels (frame based)`.

- The filter structure you select lends itself to this specification. See "Specify Time-Varying Filters" for more information.

Initial conditions

Specify the initial conditions of the filter states. To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-436.

Initial conditions on zeros side

(Not shown in dialog above.) Specify the initial conditions for the filter states on the side of the filter structure with the zeros (b_0, b_1, b_2, ...); see the diagram below.

This parameter is enabled only when the filter has both poles and zeros, *and* when you select a structure such as direct form I, which has separate filter states corresponding to the poles (a_k) and zeros (b_k). To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-436.

Initial conditions on poles side

(Not shown in dialog above). Specify the initial conditions for the filter states on the side of the filter structure with the poles (a_0, a_1, a_2, ...); see the diagram below.

This parameter is enabled only when the filter has both poles and zeros, *and* when you select a structure such as direct form I, which has separate filter states corresponding to the poles (a_k) and zeros (b_k). To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 2-436.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Note The Inherited (this choice will be removed - see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Digital Filter

View filter response

> This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined by the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note If you specify a filter in the **Filter** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

The **Data Types** pane of the Digital Filter block dialog appears as follows when **Dialog parameters** is specified in the **Coefficient source** group box. The parameters below can appear when **Dialog parameters** or **Input port(s)** is selected, depending on the filter structure and whether the coefficients are being entered via ports or on the block mask.
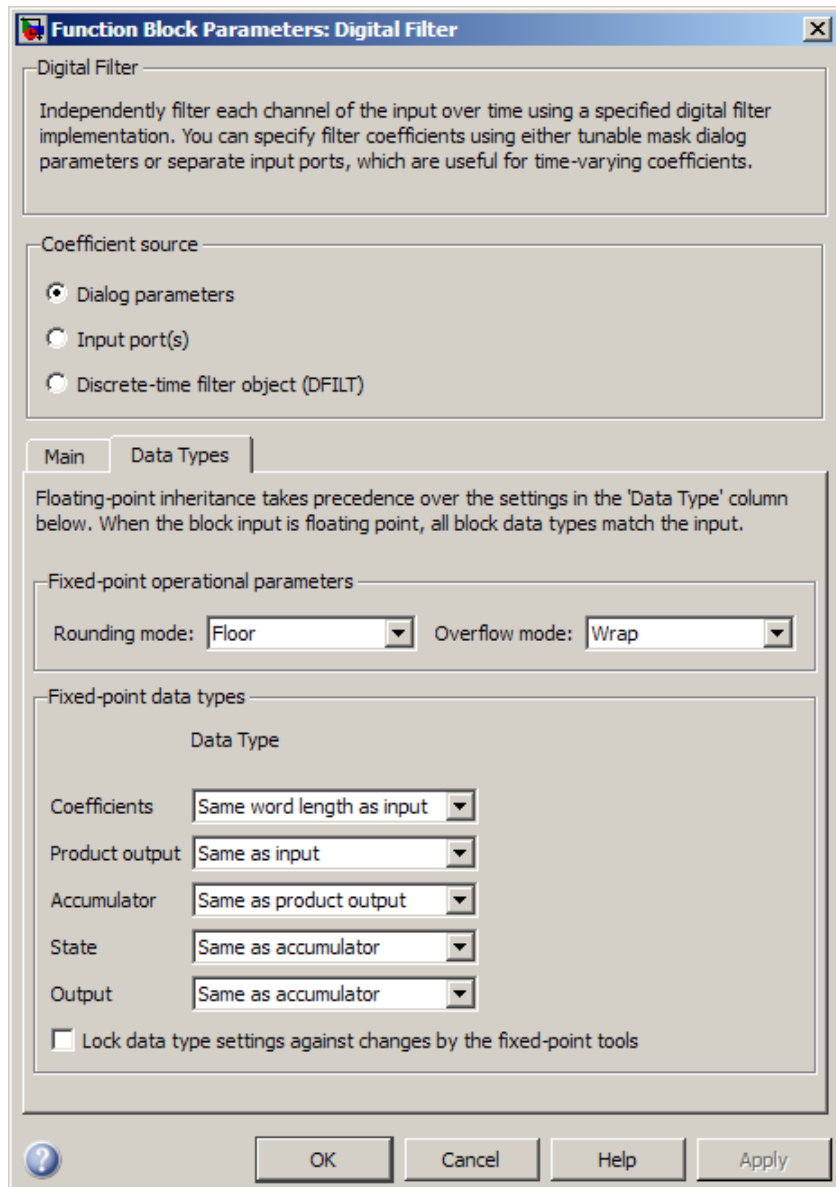
Digital Filter

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Section I/O

Choose how you specify the word length and the fraction length of the fixed-point data type going into and coming out of each section of a biquadratic filter. See "Filter Structure Diagrams" on page 2-459 for illustrations depicting the use of the section I/O data type in this block.

This parameter is only visible when the selected filter structure is biquadratic:

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

- When you select `Binary point scaling`, you can enter the word and fraction lengths of the section input and output, in bits.

- When you select `Slope and bias scaling`, you can enter the word lengths, in bits, and the slopes of the section input and output. This block requires power-of-two slope and a bias of zero.

Tap sum

Choose how you specify the word length and the fraction length of the tap sum data type of a direct form symmetric or direct form antisymmetric filter. See "Filter Structure Diagrams" on page 2-459 for illustrations depicting the use of the tap sum data type in this block.

This parameter is only visible when the selected filter structure is either `Direct form symmetric` or `Direct form antisymmetric`:

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

- When you select `Binary point scaling`, you can enter the word length and the fraction length of the tap sum accumulator, in bits.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the tap sum accumulator. This block requires power-of-two slope and a bias of zero.

Multiplicand

Choose how you specify the word length and the fraction length of the multiplicand data type of a direct form I transposed or biquadratic direct form I transposed filter. See "Filter Structure Diagrams" on page 2-459 for illustrations depicting the use of the multiplicand data type in this block.

This parameter is only visible when the selected filter structure is either `Direct form I transposed` or `Biquad direct form I transposed (SOS)`:

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

- When you select `Binary point scaling`, you can enter the word length and the fraction length of the multiplicand data type, in bits.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the multiplicand data type. This block requires power-of-two slope and a bias of zero.

Coefficients

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). See "Filter Structure Diagrams" on page 2-459 for illustrations depicting the use of the coefficient data types in this block:

- When you select `Same word length as input`, the word length of the filter coefficients match that of the input to the

block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Specify word length`, you can enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Binary point scaling`, you can enter the word length and the fraction length of the coefficients, in bits. If applicable, you can enter separate fraction lengths for the numerator and denominator coefficients.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the coefficients. If applicable, you can enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.

- The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See "Filter Structure Diagrams" on page 2-459 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block:

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

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

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

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See "Filter Structure Diagrams" on page 2-459 and "Multiplication Data Types" for illustrations depicting the use of the accumulator data type in this block:

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

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

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

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

State

Use this parameter to specify how you would like to designate the state word and fraction lengths. See "Filter Structure Diagrams" on page 2-459 for illustrations depicting the use of the state data type in this block.

This parameter is not visible for direct form and direct form I filter structures.

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

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

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

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

Output

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

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

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

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

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

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Specify Discrete-Time Filter Object

The **Main** pane of the Digital Filter block dialog appears as follows when **Discrete-time filter object (DFILT)** is specified in the **Coefficient source** group box.
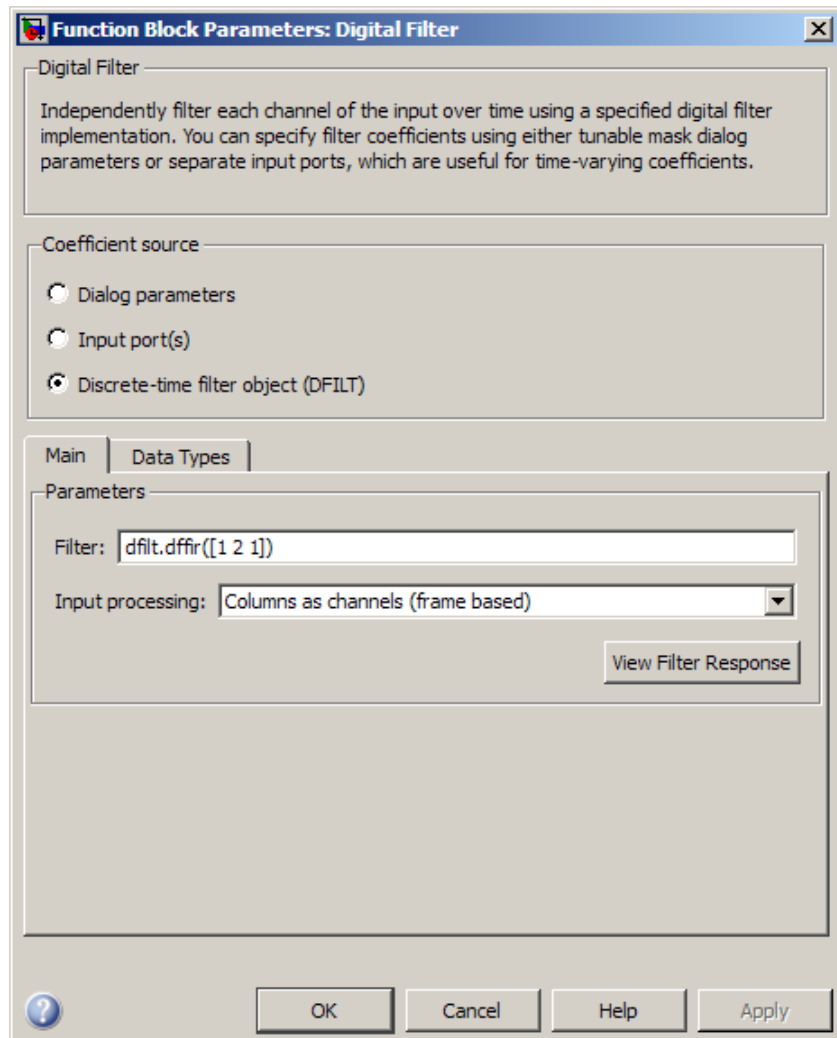
Filter

        Specify the discrete-time filter object (`dfilt`) that you would like the block to implement. You can do this in one of three ways:

- You can fully specify the `dfilt` object in the block mask, as shown in the default value.

- You can enter the variable name of a `dfilt` object that is defined in any workspace.

- You can enter a variable name for a `dfilt` object that is not yet defined.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the `dfilt` object specified in the **Filter** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

> **Note** If you specify a filter in the **Filter** parameter, you must click the **Apply** button to apply the filter before using the **View filter response** button.

The **Data Types** pane of the Digital Filter block dialog appears as follows when **Discrete-time filter object (DFILT)** is specified in the **Coefficient source** group box.

Digital Filter
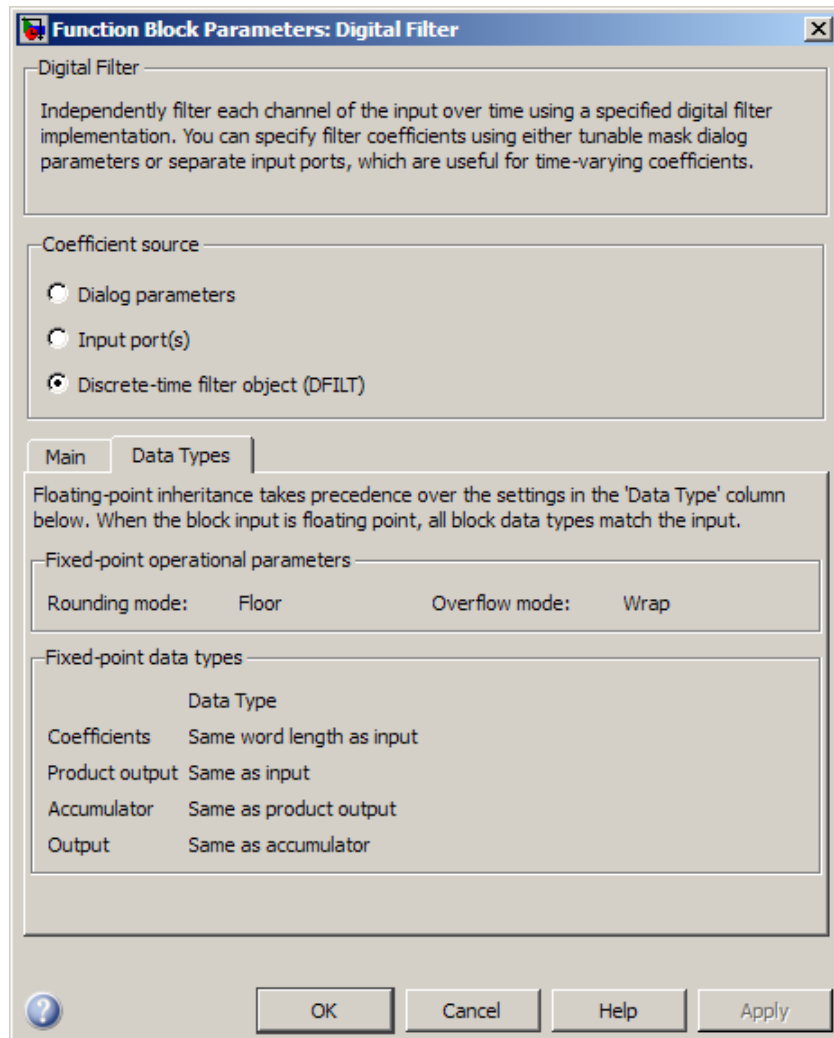
The fixed-point settings of the filter object specified on the **Main** pane are displayed on the **Data Types** pane. You cannot change these settings directly on the block mask. To change the fixed-point settings you must edit the filter object directly.

For more information on discrete-time filter objects, see the `dfilt` reference page.

Filter Structure Diagrams

The diagrams in the following sections show the filter structures supported by the Digital Filter block. They also show the data types used in the filter structures for fixed-point signals. You can set the coefficient, output, accumulator, product output, and state data types shown in these diagrams in the block dialog. This is discussed in "Dialog Box" on page 2-441.

IIR direct form I

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must be the same complexity as each other.
  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.

- When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.

- The State data type cannot be specified on the block mask for this structure, because the input and output states have the same data types as the input and output buffers.

IIR direct form I transposed

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator and denominator coefficients can be real or complex.

- Numerator and denominator coefficients must be the same complexity as each other.

- When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.

- When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- States are complex when either the input or the coefficients are complex.

- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.

Digital Filter

IIR direct form II

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Numerator and denominator coefficients must be the same complexity as each other.
  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.

- When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- States are complex when either the inputs or the coefficients are complex.

- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.

IIR direct form II transposed

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator and denominator coefficients can be real or complex.

- Numerator and denominator coefficients must be the same complexity as each other.

  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.

- When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- States are complex when either the inputs or the coefficients are complex.

- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.

Digital Filter

IIR biquadratic direct form I

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Numerator and denominator coefficients can be real or complex.

- Specify the coefficients by a M-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.

- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.

- States are complex when either the inputs or the coefficients are complex.

- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

- Scale values must have the same complexity as the coefficient SOS matrix.

- The scale value parameter must be a scalar or a vector of length M+1, where M is the number of sections.

- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and stage output data type must have the same word length but can have different fraction lengths.

The following diagram shows the data types for one section of the filter.

Digital Filter
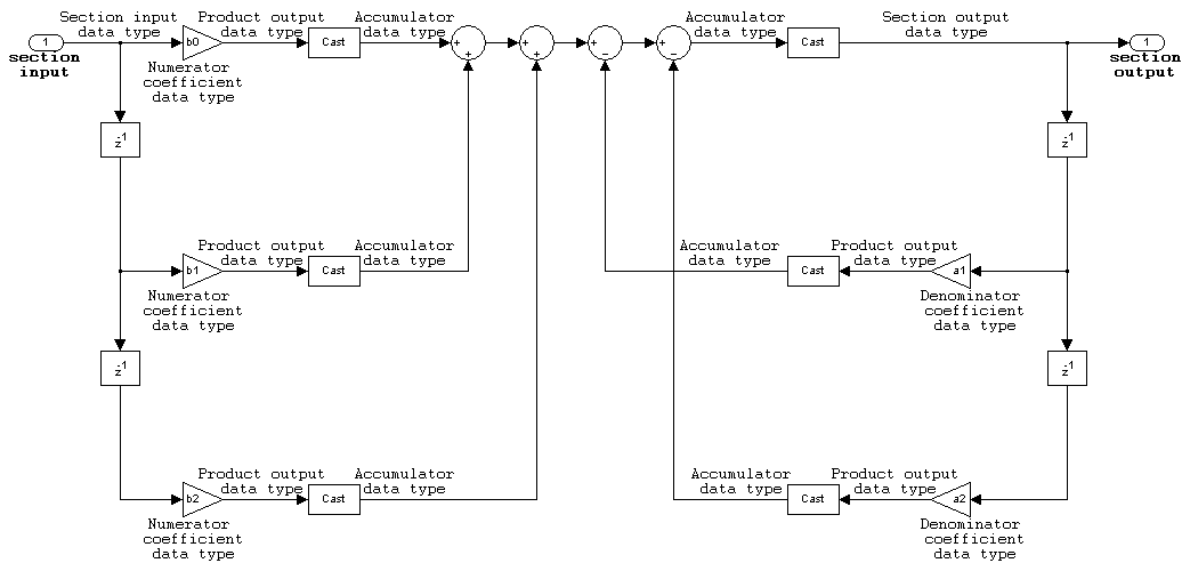
The following diagram shows the data types between filter sections.
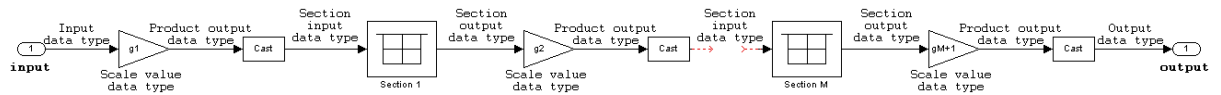
IIR biquadratic direct form I transposed
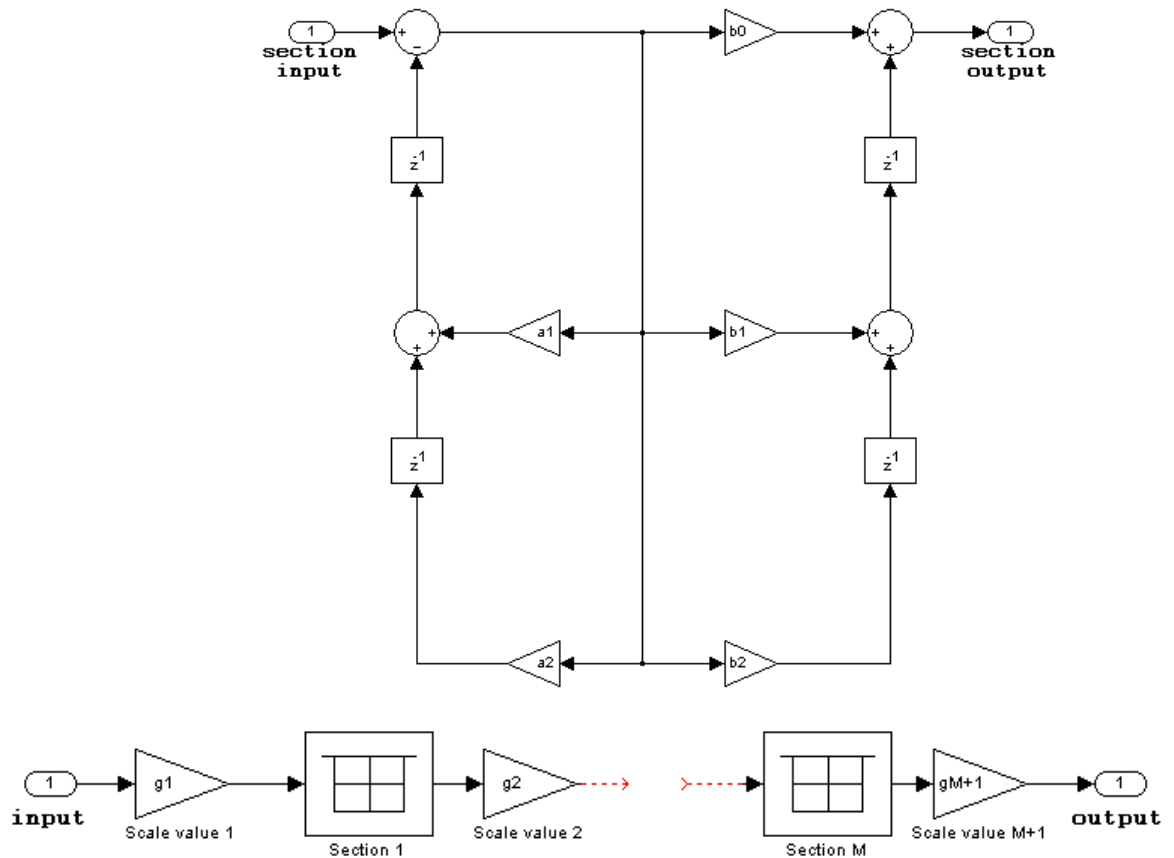
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Numerator and denominator coefficients can be real or complex.

Digital Filter

- Specify the coefficients by a M-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.

- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.

- States are complex when either the inputs or the coefficients are complex.

- Scale values must have the same complexity as the coefficient SOS matrix.

- The scale value parameter must be a scalar or a vector of length M+1, where M is the number of sections.

- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

The following diagram shows the data types for one section of the filter.

Digital Filter

The following diagram shows the data types between filter sections.

IIR biquadratic direct form II

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Numerator and denominator coefficients can be real or complex.

- Specify the coefficients by a *M*-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.

- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.

- States are complex when either the inputs or the coefficients are complex.

- Scale values must have the same complexity as the coefficient SOS matrix.

- The scale value parameter must be a scalar or a vector of length *M*+1, where *M* is the number of sections.

- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

The following diagram shows the data types for one section of the filter.

The following diagram shows the data types between filter sections.

Digital Filter

IIR biquadratic direct form II transposed

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Numerator and denominator coefficients can be real or complex.

- Specify the coefficients by a M-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.

- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.

- States are complex when either the inputs or the coefficients are complex.

- Scale values must have the same complexity as the coefficient SOS matrix.

- The scale value parameter must be a scalar or a vector of length M+1, where M is the number of sections.

- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

The following diagram shows the data types for one section of the filter.

Digital Filter

The following diagram shows the data types between filter sections.

IIR (all poles) direct form
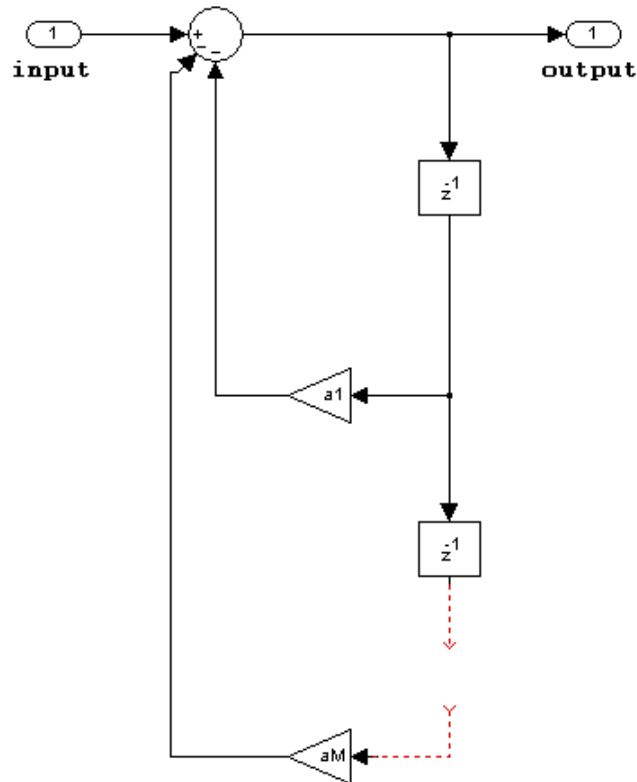
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Denominator coefficients can be real or complex.

- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

Digital Filter
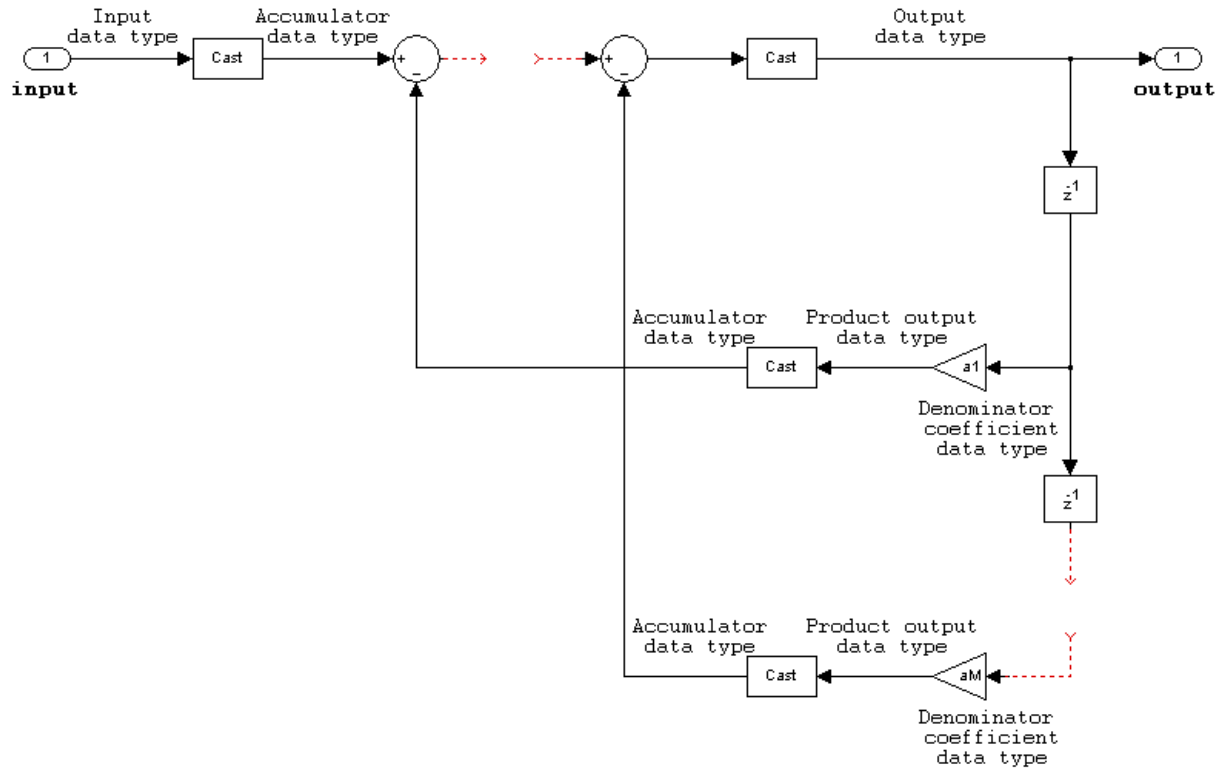
IIR (all poles) direct form transposed
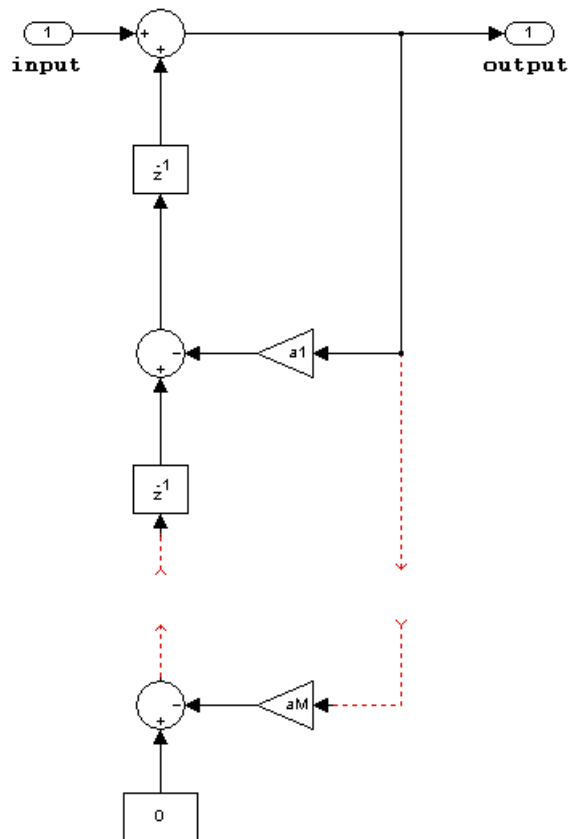
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Denominator coefficients can be real or complex.

Digital Filter
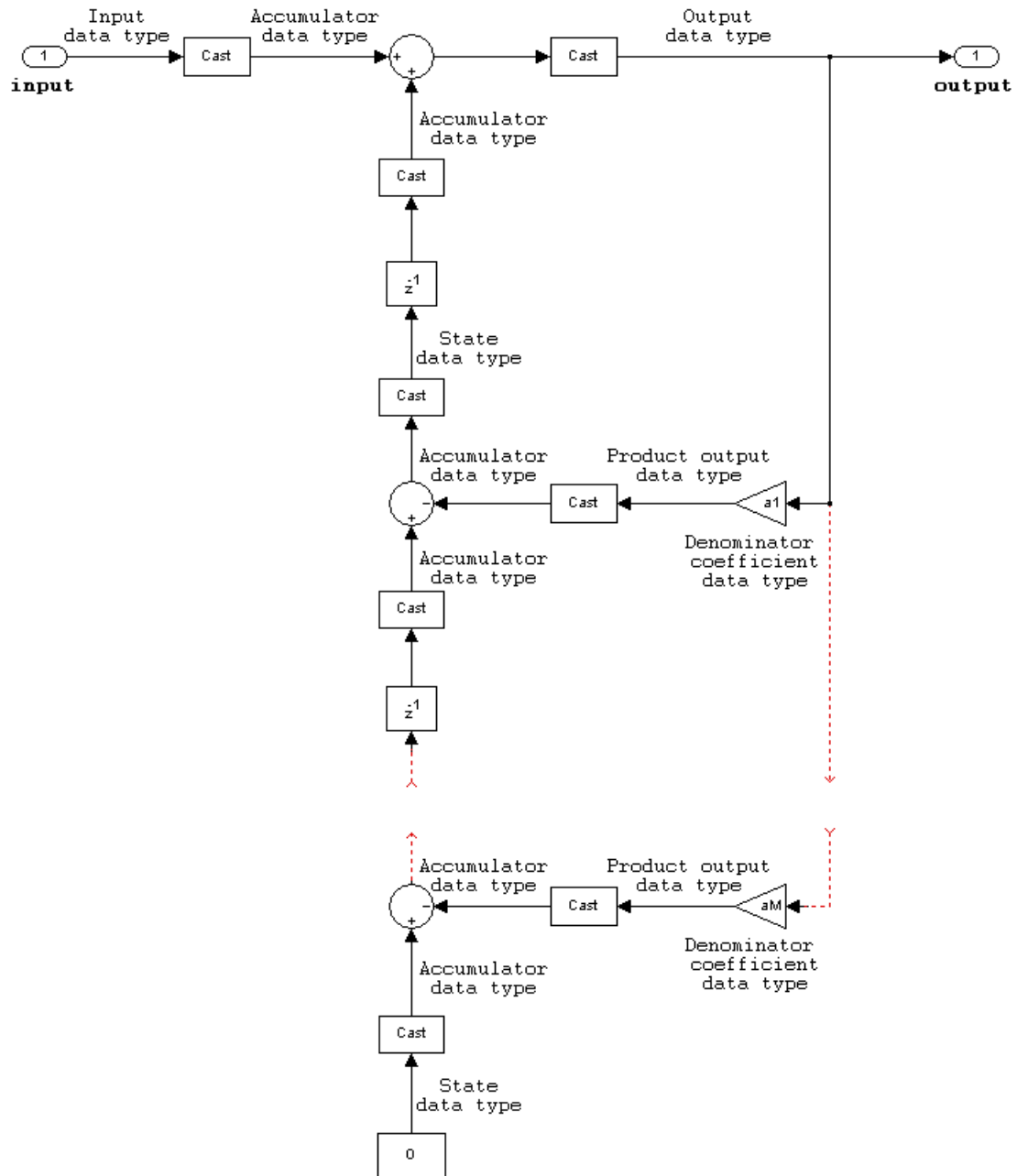
IIR (all poles) direct form lattice AR
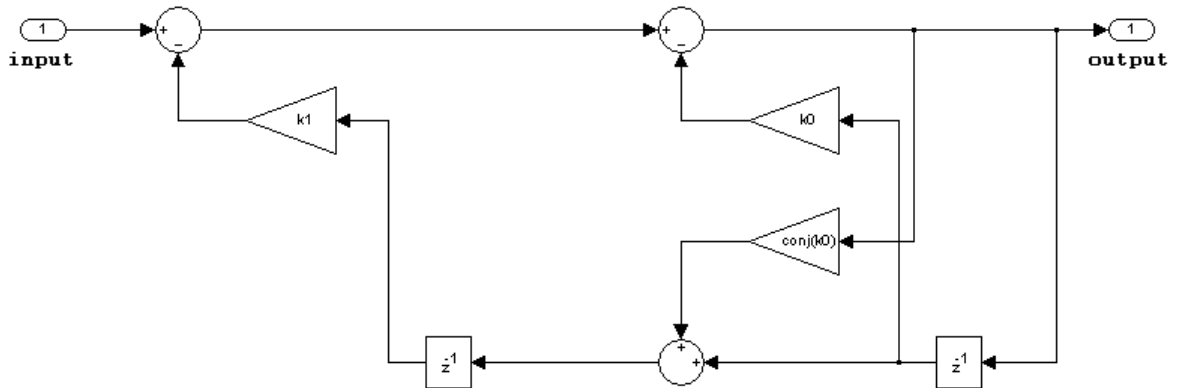
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Coefficients can be real or complex.

Digital Filter
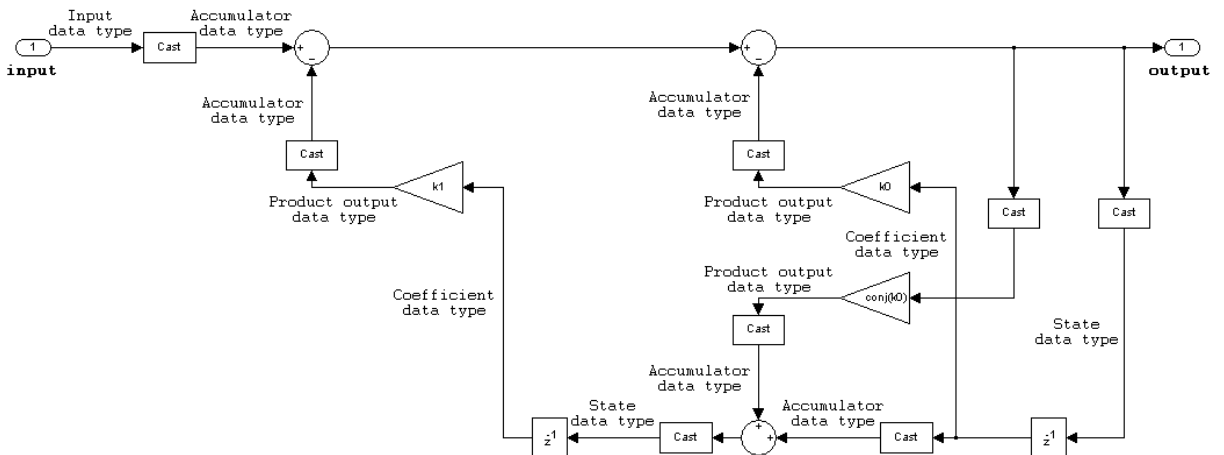
FIR (all zeros) direct form

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator coefficients can be real or complex.

- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

FIR (all zeros) direct form symmetric

Even Order — Type I   Odd Order — Type II

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator coefficients can be real or complex.

- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

- It is assumed that the filter coefficients are symmetric. Only the first half of the coefficients are used for filtering.

- The **Tap Sum** parameter determines the data type the filter uses when it sums the inputs prior to multiplication by the coefficients.

Even Order — Type I

Odd Order — Type II

Digital Filter

FIR (all zeros) direct form antisymmetric

Even Order — Type III   Odd Order — Type IV

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator coefficients can be real or complex.

- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

- It is assumed that the filter coefficients are antisymmetric. Only the first half of the coefficients are used for filtering.

- The **Tap Sum** parameter determines the data type the filter uses when it sums the inputs prior to multiplication by the coefficients.

Even Order — Type III

Odd Order — Type IV
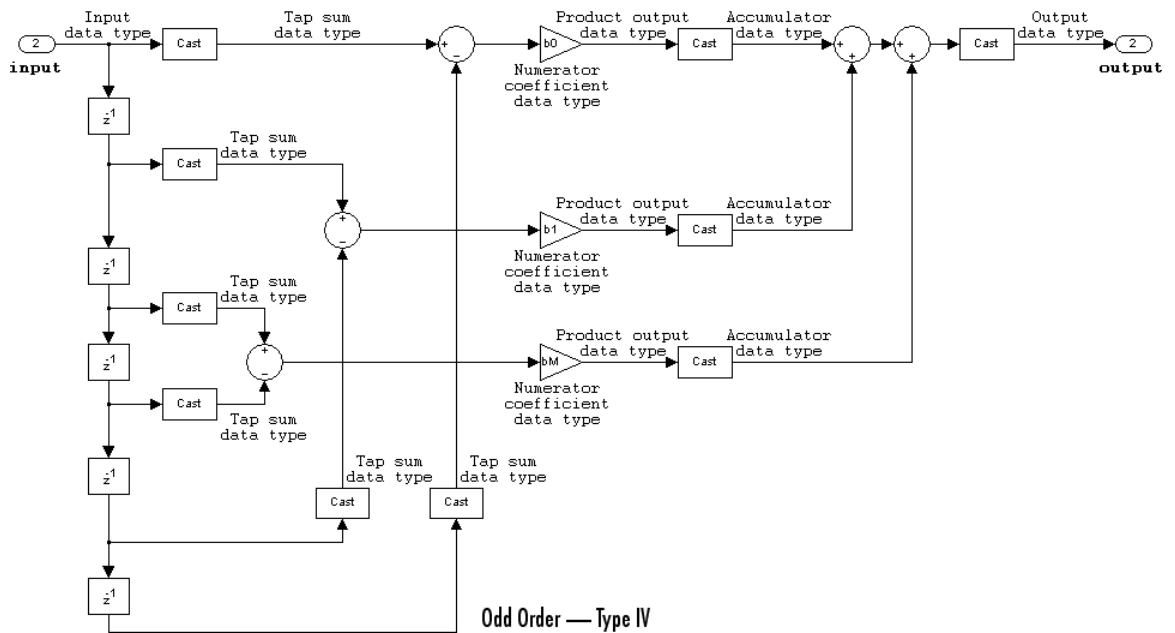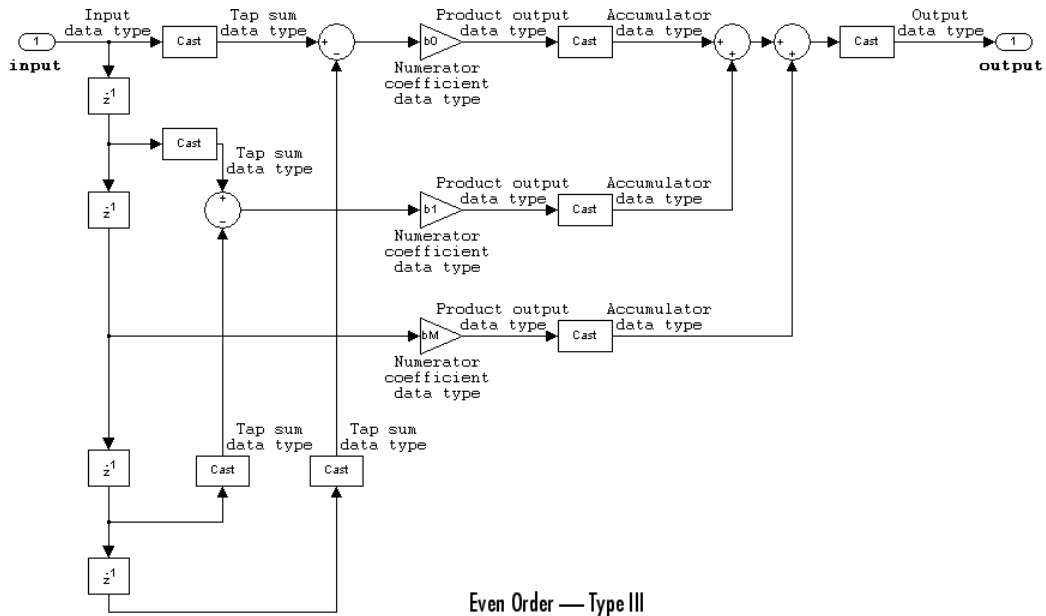
FIR (all zeros) direct form transposed
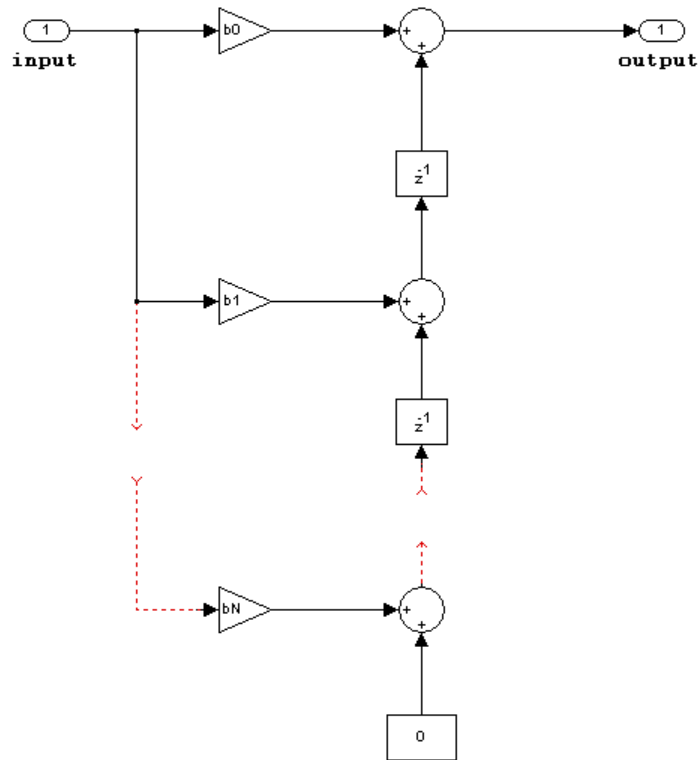
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Coefficients can be real or complex.

- States are complex when either the inputs or the coefficients are complex.
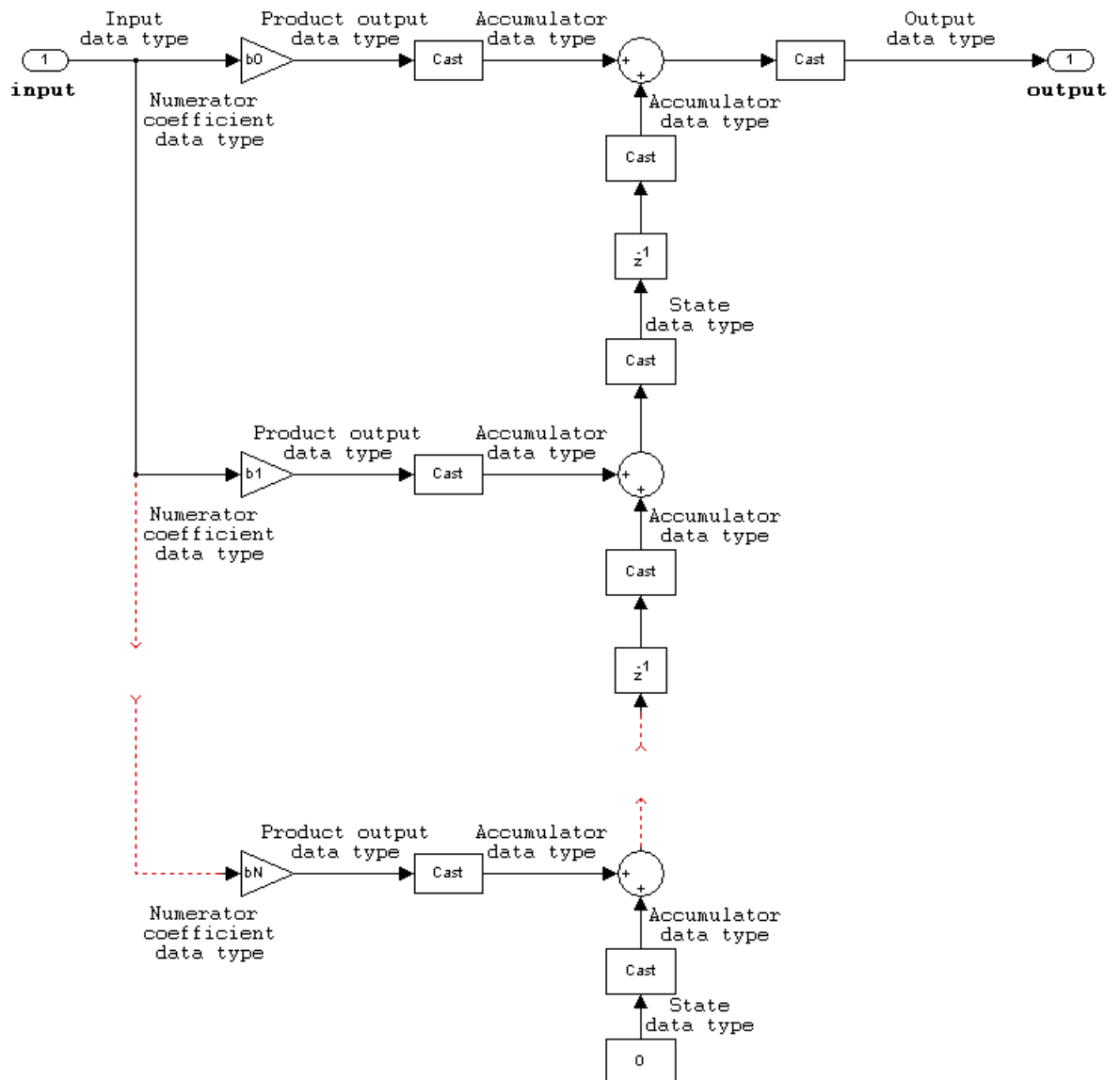
FIR (all zeros) lattice MA
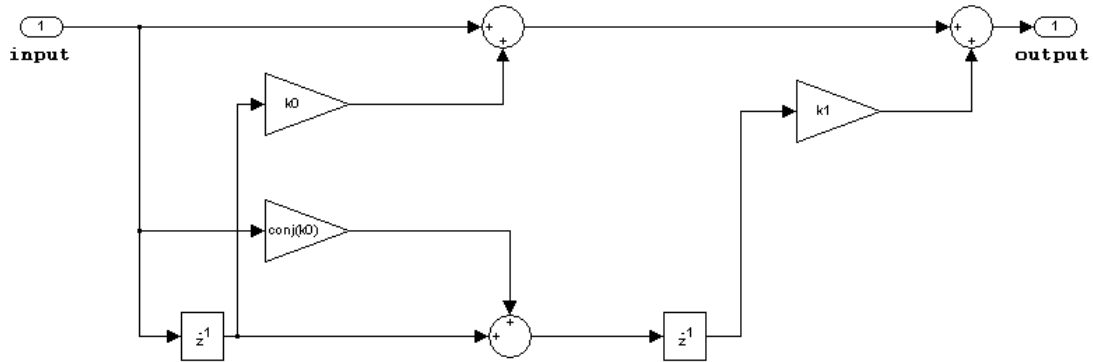
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Coefficients can be real or complex.
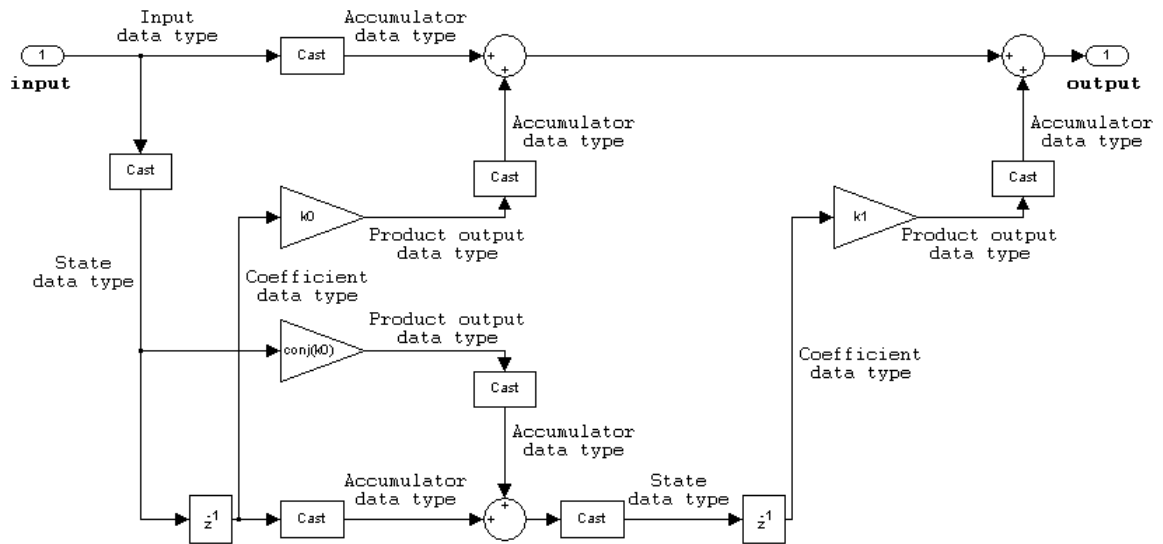
Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

Allpole Filter	DSP System Toolbox
Digital Filter Design	DSP System Toolbox
Discrete Filter	Simulink
Discrete FIR Filter	Simulink
Filter Realization Wizard	DSP System Toolbox
dfilt	DSP System Toolbox
fdatool	DSP System Toolbox

Digital Filter

`fvtool`	Signal Processing Toolbox
`sptool`	Signal Processing Toolbox

Purpose   Design and implement digital FIR and IIR filters

Library   Filtering / Filter Implementations

dsparch4

Description
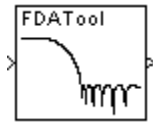
Note Use this block to design, analyze, and efficiently implement floating-point filters. The following blocks also implement digital filters, but serve slightly different purposes:

- Digital Filter — Use to efficiently implement floating-point or fixed-point filters that you have already designed. This block provides the same exact filter implementation as the Digital Filter Design block.

- Filter Realization Wizard — Use to implement floating-point or fixed-point filters built from Sum, Gain, and Unit Delay blocks. You can either design the filter within this block, or import the coefficients of a filter that you designed elsewhere.

The Digital Filter Design block implements a digital FIR or IIR filter that you design using the Filter Design and Analysis Tool (fdatool) GUI. This block provides the same filter implementation as the Digital Filter block.

You must specify whether the block performs frame-based or sample-based processing on the input by setting the **Input processing** parameter. The block applies the specified filter to each channel of a discrete-time input signal, and outputs the result. The outputs of the block numerically match the outputs of the Digital Filter block, the MATLAB filter function, and the DSP System Toolbox filter function.

The sampling frequency, Fs, that you specify in the FDATool GUI should be identical to the sampling frequency of the input to the Digital Filter Design block. When the sampling frequencies do not match, the

Digital Filter Design

Digital Filter Design block returns a warning message and inherits the sampling frequency of the input block.

Designing the Filter

Double-click the Digital Filter Design block to open FDATool. Use FDATool to design or import a digital FIR or IIR filter. To learn how to design filters with this block and FDATool, see the following topics:

- "Digital Filter Design Block"
- `fdatool` reference page

Tuning the Filter During Simulation

You can tune the filter specifications in FDATool during simulations as long as your changes do not modify the filter length or filter order. The filter updates as soon as you apply any filter changes in FDATool.

Examples   See the "Digital Filter Design Block" section in the DSP System Toolbox documentation.
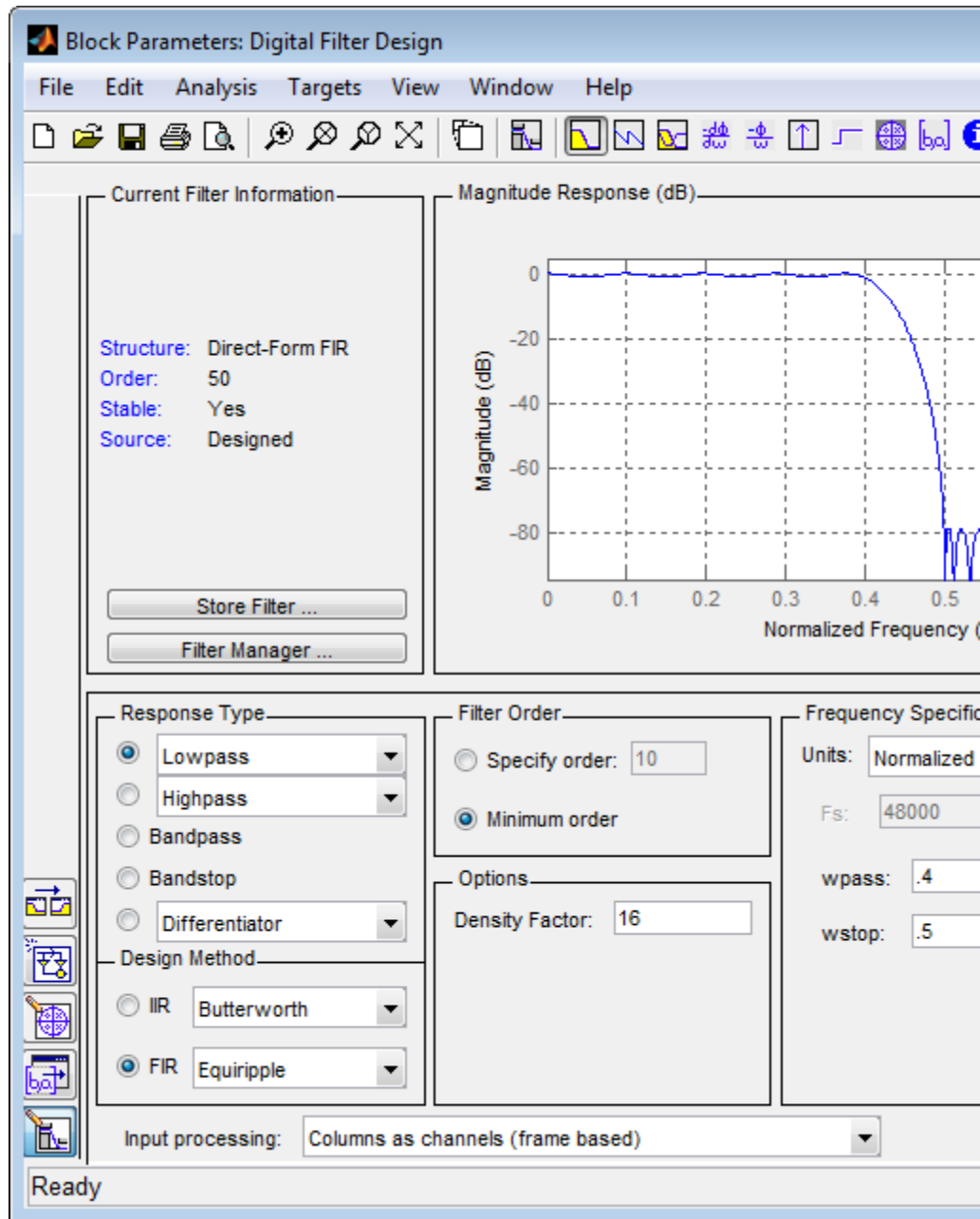
**Dialog
Box**

For more information about the parameters on this dialog, see
"FDATool".

Digital Filter Design

Supported Data Types
- Double-precision floating point
- Single-precision floating point

See Also

Analog Filter Design	DSP System Toolbox
Window Function	DSP System Toolbox
fdatool	DSP System Toolbox
filter	DSP System Toolbox
fvtool	Signal Processing Toolbox
sptool	Signal Processing Toolbox

To learn how to use this block and FDATool, see the following:

- "Filter Design"
- "Filter Analysis"
- "Digital Filter Design Block"

Purpose   Design and implement a variety of FIR filters

Library   dspobslib

Description

Note The Digital FIR Filter Design block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Digital FIR Filter Design block designs a discrete-time (digital) FIR filter in one of several different band configurations using a window method. Most of these filters are designed using the Signal Processing Toolbox fir1 function, and are real with linear phase response. The block applies the filter to a discrete-time input using the Direct-Form II Transpose Filter (Obsolete) 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.

Digital FIR Filter Design (Obsolete)
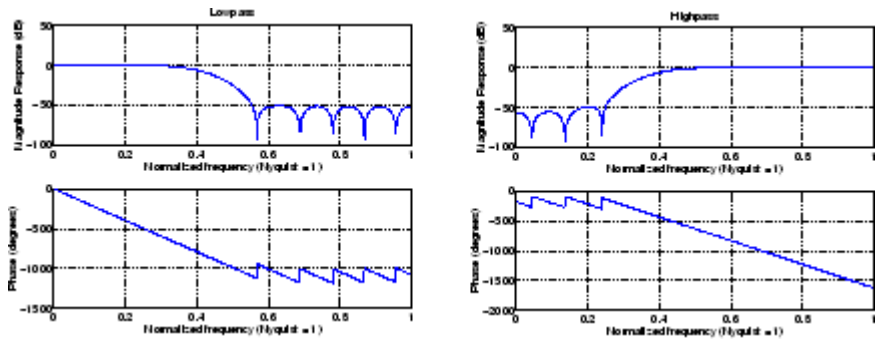
- **Bandpass** and **Bandstop**

  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.
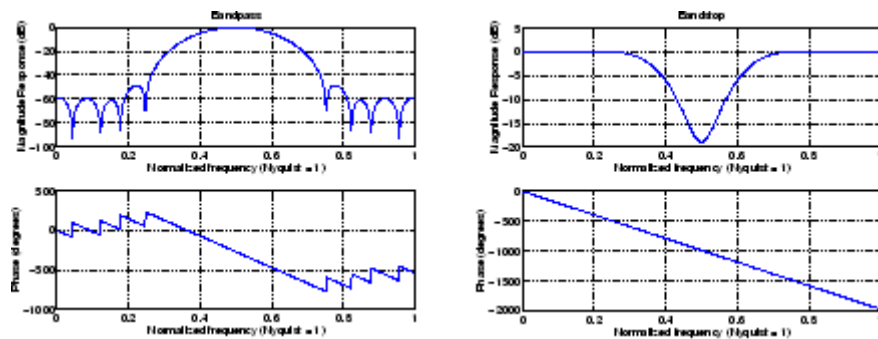
- **Multiband**

  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]
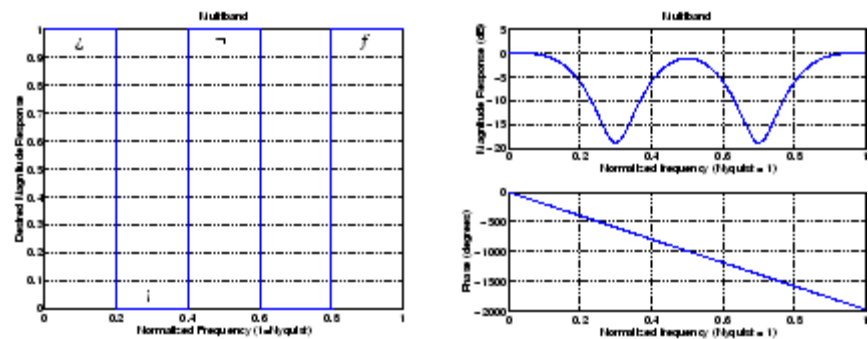
- **Arbitrary shape**

  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 Signal Processing Toolbox `fir2` function.)

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

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

Duplicate frequencies can be used to specify a step in the response (such as band 2 below). The figure shows an order-100 filter with five bands.

Frequency = [0.0 0.3 0.3 0.6 0.7 1.0]

Gains = [0.0 1.0 0.0 0.0 0.5 0.5]

Band: ¿ i ¬ f
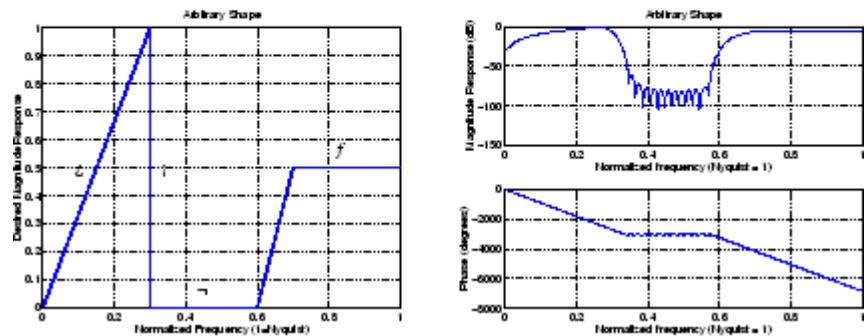
The **Window type** parameter allows you to select from a variety of different windows. See the Window Function block reference for a complete description of the available options.

Dialog Box
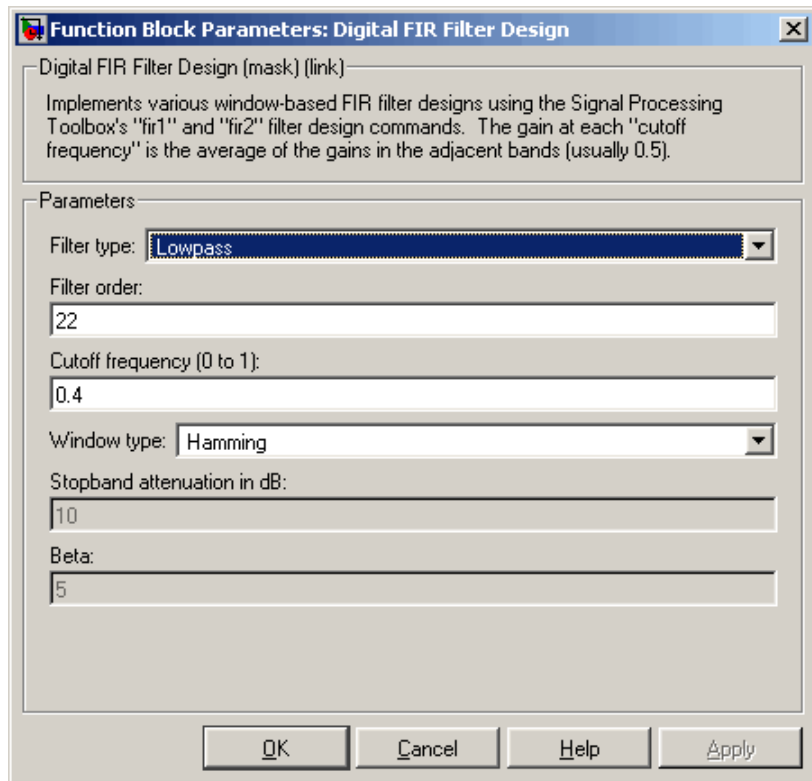
Function Block Parameters: Digital FIR Filter Design

Digital FIR Filter Design (mask) (link)

Implements various window-based FIR filter designs using the Signal Processing Toolbox's "fir1" and "fir2" filter design commands. The gain at each "cutoff frequency" is the average of the gains in the adjacent bands (usually 0.5).

Parameters

Filter type: Lowpass

Filter order:

22

Cutoff frequency (0 to 1):

0.4

Window type: Hamming

Stopband attenuation in dB:

10

Beta:

5

| OK | Cancel | Help | Apply |

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.

Digital FIR Raised Cosine Filter Design (Obsolete)

Purpose   Design and implement raised cosine FIR filter

Library   dspobslib

Description

firrcos

Note The Digital FIR Raised Cosine Filter Design block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Digital FIR Raised Cosine Filter Design block uses the Signal Processing Toolbox `firrcos` function 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 (Obsolete) block.

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

The frequency response of the raised cosine filter is

$$H(f) = \begin{cases} \dfrac{1}{2f_{n0}} & 0 \le |f| \le (1-R)f_{n0} \\[2em] \left[\dfrac{1 + \cos\dfrac{\pi}{2Rf_{n0}}|f| - (1-R)f_{n0}}{4f_{n0}} \right] & (1-R)f_{n0} \le |f| \le (1+R)f_{n0} \\[2em] 0 & (1+R)f_{n0} \le |f| \le 1 \end{cases}$$

where $H(f)$ is the magnitude response at frequency f, f_{n0} is the normalized cutoff frequency (-6 dB) specified by the **Upper cutoff frequency** parameter, and R is a rolloff factor in the range `[0,1]` determining the passband-to-stopband transition width.

The **Square-root raised cosine filter** option designs a filter with magnitude response $\sqrt{H(f)}$. This is useful when the filter is part of a pair of matched filters.

When the **Design method** parameter is set to **Rolloff factor**, the secondary **Rolloff factor** parameter is enabled, and R can be directly specified. When **Design method** is set to **Transition bandwidth**, the secondary **Transition bandwidth** parameter is enabled, and the transition region bandwidth, Δf, can be specified in place of R. The transition region is centered on f_{n0} and must be sufficiently narrow to satisfy

$$0 < \left(f_{n0} \pm \frac{\Delta f}{2} \right) < 1$$

The **Upper cutoff frequency** and **Transition bandwidth** parameter values are normalized to half the sample frequency.

The **Window type** parameter allows you to apply a variety of different windows to the raised cosine filter. See the Window Function block reference for a complete description of the available options.

Algorithm   The filter output is computed by convolving the input with a truncated, delayed, windowed version of the filter's impulse response. The impulse response for the raised cosine filter is

$$h(kT_s) = \frac{1}{F_s} \mathrm{sinc}(2\pi k T_s f_{n0}) \frac{\cos(2\pi R k T_s f_{n0})}{1 - (4 R k T_s f_{n0})^2} \quad -\infty < k < \infty$$

which has limits

$$h(0) = \frac{1}{F_s}$$

and

$$h\left(\pm\frac{1}{4Rf_{n0}}\right) = \frac{R}{2F_s}\sin\left(\frac{\pi}{2R}\right)$$

The impulse response for the square-root raised cosine filter is

$$h(kT_s) = \frac{4R\cos((1+R)2\pi kT_sf_{n0}) + \dfrac{\sin((1-R)2\pi kT_sf_{n0})}{8RkT_sf_{n0}}}{\pi F_s\sqrt{\dfrac{1}{2f_{n0}}}((8RkT_sf_{n0})^2 - 1)} \quad -\infty < k < \infty$$

which has limits

$$h(0) = (-4R - \pi + \pi R)\frac{\sqrt{2f_{n0}}}{\pi F_s}$$

and

$$\left(\pm\frac{1}{8Rf_{n0}}\right) = \frac{\sqrt{2f_{n0}}}{2\pi F_s}\left[\pi(1+R)\sin\left(\frac{\pi(1+R)}{4R}\right) - 4R\sin\left(\frac{\pi(R-1)}{4R}\right)\right.$$
$$\left. +\pi(R-1)\cos\left(\frac{\pi(R-1)}{4R}\right)\right]$$

**Dialog
Box**

Function Block Parameters: Digital FIR Raised Cosine Filter Design

Digital FIR Raised Cosine Filter Design (mask) (link)

Linear phase digital FIR lowpass raised cosine filter.

Parameters

Filter order:

63

Upper cutoff frequency (0 to 1):

0.5

☑ Square-root raised cosine filter

Design method: Rolloff factor

Rolloff factor (0 to 1):

0.6

Window type: Boxcar

Stopband attenuation in dB:

5

Beta:

10

Initial conditions:

0

OK   Cancel   Help   Apply

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.

Digital FIR Raised Cosine Filter Design (Obsolete)

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 (Obsolete) 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.

Purpose   Design and implement IIR filter

Library   dspobslib
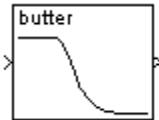
Description

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 (Obsolete) 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.

Digital IIR Filter Design (Obsolete)

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

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

The table below lists the available parameters for each design/band combination. For lowpass and highpass band configurations, these parameters include the passband edge frequency f_{np}, the stopband edge frequency f_{ns}, the passband ripple R_p, and the stopband attenuation R_s. For bandpass and bandstop configurations, the parameters include the lower and upper passband edge frequencies, f_{np1} and f_{np2}, the lower and upper stopband edge frequencies, f_{ns1} and f_{ns2}, the passband ripple R_p, and the stopband attenuation R_s. Frequency values are normalized to half the sample frequency, and ripple and attenuation values are in dB.

	Lowpass	Highpass	Bandpass	Bandstop
Butterworth	Order, f_{np}	Order, f_{np}	Order, f_{np1}, f_{np2}	Order, f_{np1}, f_{np2}
Chebyshev Type I	Order, f_{np}, R_p	Order, f_{np}, R_p	Order, f_{np1}, f_{np2}, R_p	Order, f_{np1}, f_{np2}, R_p
Chebyshev Type II	Order, f_{ns}, R_s	Order, f_{ns}, R_s	Order, f_{ns1}, f_{ns2}, R_s	Order, f_{ns1}, f_{ns2}, R_s
Elliptic	Order, f_{np}, R_p, R_s	Order, f_{np}, R_p, R_s	Order, f_{np1}, f_{np2}, R_p, R_s	Order, f_{np1}, f_{np2}, R_p, R_s

The digital filters are designed using Signal Processing Toolbox software's filter design commands `butter`, `cheby1`, `cheby2`, and `ellip`.

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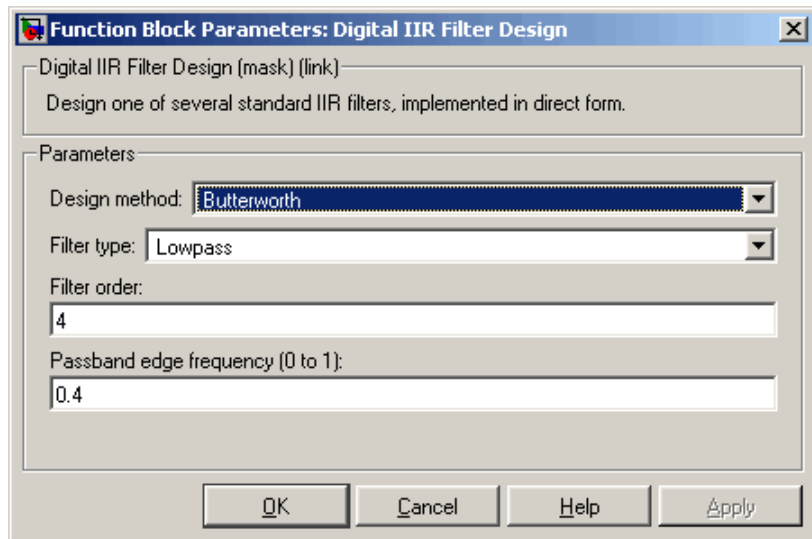
The parameters displayed in the dialog box vary for different
design/band combinations. Only some of the parameters listed below
are visible in the dialog box at any one time.

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.

Digital IIR Filter Design (Obsolete)

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.

Purpose   Apply IIR filter to input

Library   dspobslib
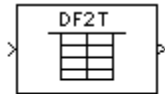
Description

Direct-Form II
Transpose Filter

Note This block is now just an implementation of the Digital Filter block.

Discrete Impulse

Purpose   Generate discrete impulse

Library   Sources

dspsrcs4

Description   The Discrete Impulse block generates an impulse (the value 1) at output
sample D+1, where D is specified by the **Delay** parameter (D ≥0). All
output samples preceding and following sample D+1 are zero.

When D is a length-*N* vector, the block generates an *M*-by-*N* matrix
output representing *N* distinct channels, where frame size *M* is
specified by the **Samples per frame** parameter. The impulse for the
ith channel appears at sample D(i)+1. For *M*=1, the output is sample
based; otherwise, the output is frame based.

The **Sample time** parameter value, T_s, specifies the output signal
sample period. The resulting frame period is $M*T_s$.

Examples   Construct the model below.

Configure the Discrete Impulse block to generate a frame-based
three-channel output of type double, with impulses at samples 1, 4, and
6 of channels 1, 2, and 3, respectively. Use a sample period of 0.25 and a
frame size of 4. The corresponding settings should be as follows:

- **Delay** = [0 3 5]
- **Sample time** = 0.25
- **Samples per frame** = 4
- **Output data type** = double

Run the model and look at the output, `dsp_examples_yout`. The first few samples of each channel are shown below.

```
dsp_examples_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).

Discrete Impulse

The **Main** pane of the Discrete Impulse block dialog appears as follows.
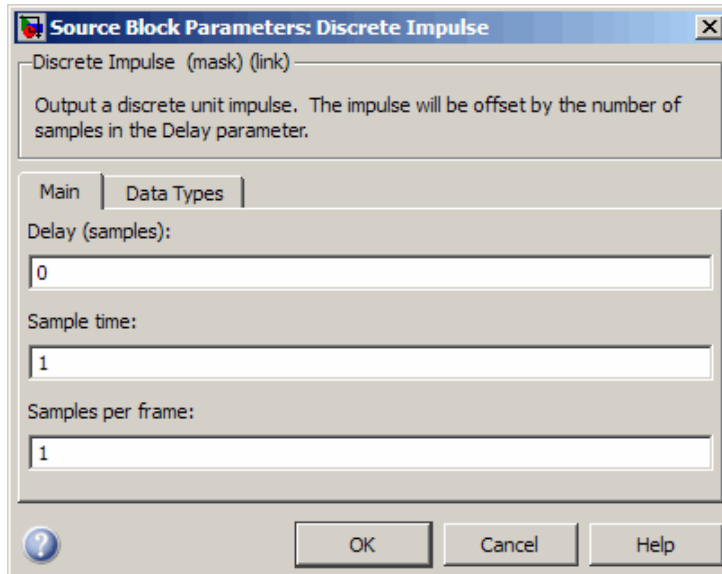
Delay

    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. When the value of this parameter is 1, the block outputs a sample-based signal.

The **Data Types** pane of the Discrete Impulse block dialog appears as follows.
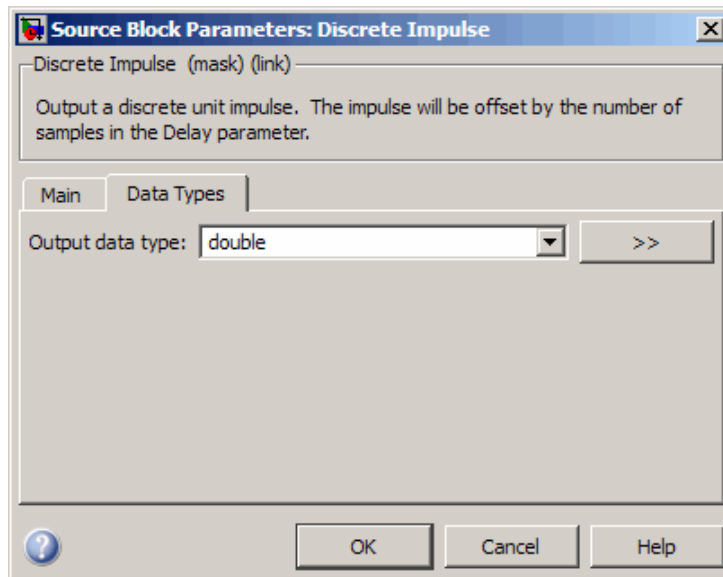
Source Block Parameters: Discrete Impulse

Discrete Impulse (mask) (link)

Output a discrete unit impulse. The impulse will be offset by the number of samples in the Delay parameter.

Main | Data Types

Output data type: double | >>

OK | Cancel | Help

Output data type

Specify the output data type for this block. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`. When you select this option, the output data type and scaling matches that of the next downstream block.

- A built in data type, such as `double`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

Discrete Impulse

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

Data Type Conversion	Simulink
Constant	Simulink
Multiphase Clock	DSP System Toolbox
N-Sample Enable	DSP System Toolbox
Signal From Workspace	DSP System Toolbox
impz	Signal Processing Toolbox

Display

Purpose   Show value of input

Library   Sinks

dspsnks4

Description   The Display block is an implementation of the Simulink Display block. See Display for more information.

Downsample

Purpose　　　Resample input at lower rate by deleting samples

Library　　　Signal Operations

dspsigops

Description

The Downsample block decreases the sampling rate of the input by deleting samples. When the block performs frame-based processing, it resamples the data in each column of the M_i-by-N input matrix independently. When the block performs sample-based processing, it treats each element of the input as a separate channel and resamples each channel of the input array across time. The resample rate is K times lower than the input sample rate, where K is the integer you specify for the **Downsample factor** parameter. The Downsample block resamples the input by discarding $K–1$ consecutive samples following each sample that is passed through to the output.

The **Sample offset** parameter delays the output samples by an integer number of sample periods, D, where $0 \leq D \leq (K–1)$, so that any of the K possible output phases can be selected. For example, when you downsample the sequence 1, 2, 3, ... by a factor of 4, you can select from the following four phases.

Input Sequence	Sample Offset, D	Output Sequence (K=4)
1,2,3,...	0	1,5,9,13,17,21,25,29,...
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 of the latter three output sequences in the table is a result of the default zero **Initial conditions** parameter setting for this example. See "Latency" on page 2-527 for more information on the **Initial conditions** parameter.

This block supports triggered subsystems when you set the **Rate options** parameter to `Enforce single-rate processing`.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the input can be an N-D array. The Downsample block treats each element of the input as a separate channel, and resamples each channel of the input over time. The block downsamples the input array by discarding $K-1$ samples following each sample that it passes through to the output. The input and output sizes of the Downsample block are identical.

The **Rate options** parameter specifies how the block adjusts the rate at the output to accommodate the reduced number of samples. There are two available options:

- `Allow multirate processing`

  When you select `Allow multirate processing`, the sample period of the output is K times longer than the input sample period ($T_{so} = KT_{si}$).

- `Enforce single-rate processing`

  When you select `Enforce single-rate processing`, 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. In this mode, the behavior of the block is similar to the operation of a Sample and Hold block with a repeating trigger event of period KT_{si}.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each of the N input columns as an individual channel containing M_i sequential time samples. 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.

Downsample

The **Rate options** parameter specifies how the block adjusts the rate at the output to accommodate the reduced number of samples. There are two available options:

- `Allow multirate processing`

  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 ex_downsample_ref1 model 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.
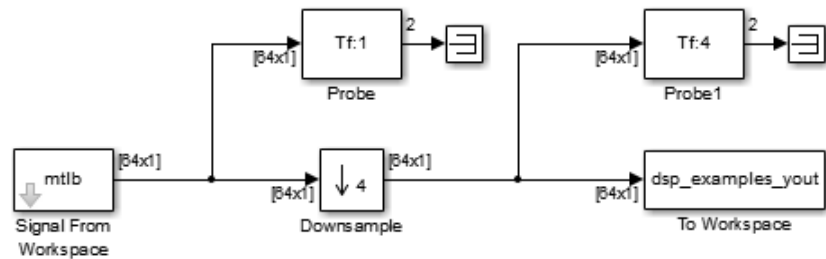
- `Enforce single rate processing`

  The block generates the output at the slower (downsampled) rate 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 ex_downsample_ref2 model shows a single-channel input with a frame size of 64 being downsampled by a factor of 4 to a frame size of 16. The input and output frame rates are identical.
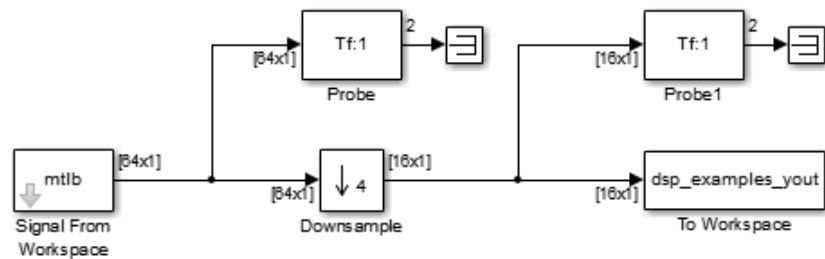
Latency

The Downsample block has *zero-tasking latency* in the following cases:

- The **Downsample factor** parameter, *K*, is 1.

- The **Input processing** parameter is set to Columns as channels (frame based) and the **Rate options** parameter is set to Enforce single-rate processing.

- The **Input processing** parameter is set to Columns as channels (frame based) and the input frame size is equal to one.

- The **Input processing** parameter is set to Elements as channels (sample based), and **Sample offset** parameter, *D*, is 0.

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+2*K*, and so on. When there is zero-tasking latency, the block ignores the value of the **Initial conditions** parameter.

In all other cases, the latency is nonzero:

- When the **Input processing** parameter is set to Elements as channels (sample based), the latency is one sample.

- When the **Input processing** parameter is set to Columns as channels (frame based) and the input frame size is greater than one, the latency is one frame.

Downsample

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 conditions** parameter can be an array of the same size as the input, 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 conditions** value can be an M_i-by-N matrix containing one value for each channel, or a scalar to be repeated across all elements of the M_i-by-N matrix. See the following example for an illustration of this case.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

Examples

Open the ex_downsample_ref3 model.
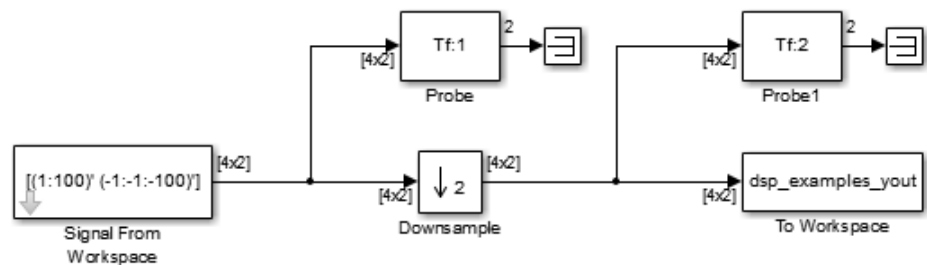
Run the model and look at the output, yout. The first few samples of each channel are as follows:

yout =

```
11    -11
12    -12
13    -13
14    -14
 2     -2
 4     -4
 6     -6
 8     -8
10    -10
12    -12
14    -14
```

You can see from the two Probe blocks that there are at least two distinct frame rates in this model. Because you ran this model in multirate, 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 (row D+1, where D is the **Sample offset**) appears in the output as sample 5 (sample M_i+1, where M_i is the input frame size).

Downsample

Function Block Parameters: Downsample

Downsample (mask) (link)

Downsample by an integer factor.

Parameters

Downsample factor, K:

2

Sample offset (0 to K-1):

0

Input processing: Columns as channels (frame based)

Rate options: Enforce single-rate processing

OK Cancel Help Apply
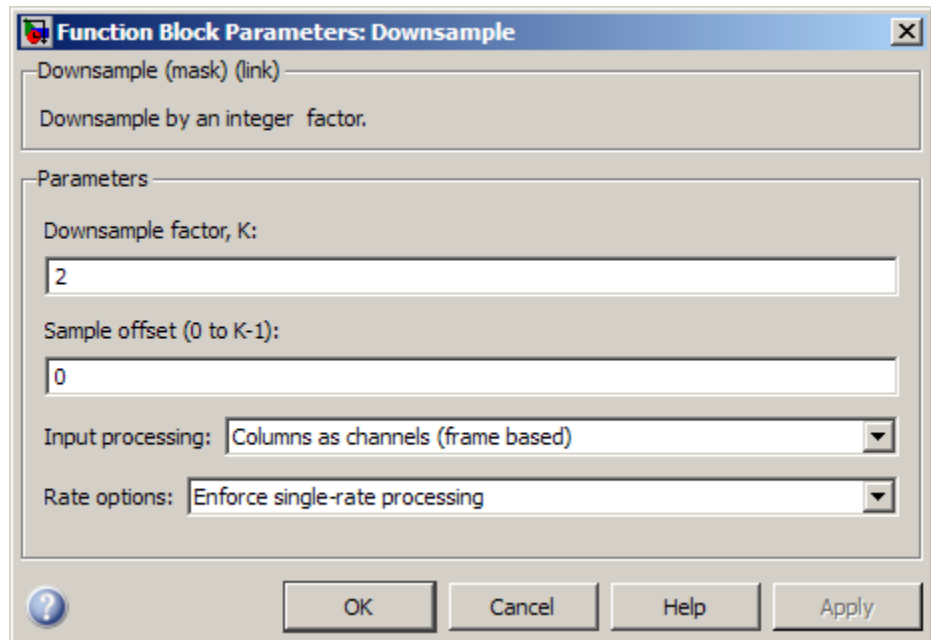
Downsample factor
> The integer factor, K, by which to decrease the input sample rate.

Sample offset
> The sample offset, D, which must be an integer in the range [0, K–1].

Input processing
> Specify how the block should process the input. You can set this parameter to one of the following options:
>
> - Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel. See "Frame-Based Processing" on page 2-525 for more information
>
> - Elements as channels (sample based) — When you select this option, the block treats each element of the input as a

separate channel. See "Sample-Based Processing" on page 2-525 for more information.

Note The Inherited (this choice will be removed - see release notes) option will be removed in a future release.

Rate options

Specify the method by which the block should downsample the input. You can select one of the following options:

- Enforce single-rate processing — When you select this option, the block maintains the input sample rate.

- Allow multirate processing — When you select this option, the block downsamples the signal such that the output sample rate is *K* times slower than the input sample rate.

Initial conditions

The value with which the block is initialized for cases of nonzero latency. You can specify a scalar, or an array of the same size as the input. This parameter appears only when you set the **Rate options** parameter to Allow multirate processing.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers

Downsample

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

FIR Decimation	DSP System Toolbox
FIR Rate Conversion	DSP System Toolbox
Repeat	DSP System Toolbox
Sample and Hold	DSP System Toolbox
Upsample	DSP System Toolbox

Purpose   Generate discrete- or continuous-time constant signal

Library   Sources

dspobslib

Description

> **Note** The DSP Constant block is still supported but is likely to be
> obsoleted in a future release. We strongly recommend replacing this
> block with the Constant block.

The DSP Constant block generates a signal whose value remains
constant throughout the simulation. The **Constant value** parameter
specifies the constant to output, and can be any valid MATLAB
expression that evaluates to a scalar, vector, or matrix.

When **Sample mode** is set to Continuous, the output is a
continuous-time signal. When **Sample mode** is set to Discrete, the
Sample time parameter is visible, and the signal has the discrete
output period specified by the **Sample time** parameter.

You can set the output signal to Frame-based, Sample-based, or
Sample-based (interpret vectors as 1-D) with the **Output**
parameter.

DSP Constant (Obsolete)

Dialog Box

The **Main** pane of the DSP Constant block dialog box appears as follows.
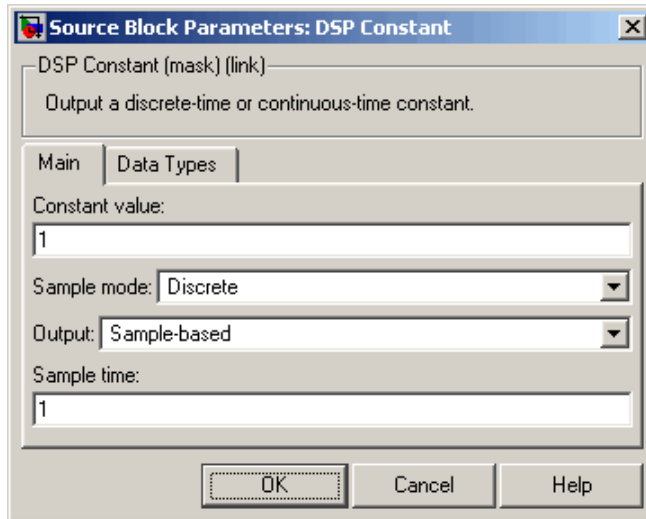
Constant value

Specify the constant to generate. This parameter is Tunable; values entered here can be tuned, but their dimensions must remain fixed.

When you specify any data type information in this field, it is overridden by the value of the **Output data type** parameter in the **Data Types** pane, unless you select Inherit from 'Constant value'.

Sample mode

Specify the sample mode of the output, Discrete for a discrete-time signal or Continuous for a continuous-time signal.

Output

Specify whether the output is Sample-based (interpret vectors as 1-D), Sample-based, or Frame-based. When you select Sample-based and the output is a vector, its dimension

is constrained to match the **Constant value** dimension (row or
column). When you select Sample-based (interpret vectors
as 1-D), however, the output has no specified dimensionality.

Sample time
    Specify the discrete sample period for sample-based outputs.
    When you select Frame-based for the **Output** parameter, this
    parameter is named **Frame period**, and is the discrete frame
    period for the frame-based output. This parameter is only visible
    when you select Discrete for the **Sample mode** parameter.

The **Data Types** pane of the DSP Constant block dialog box appears
as follows.
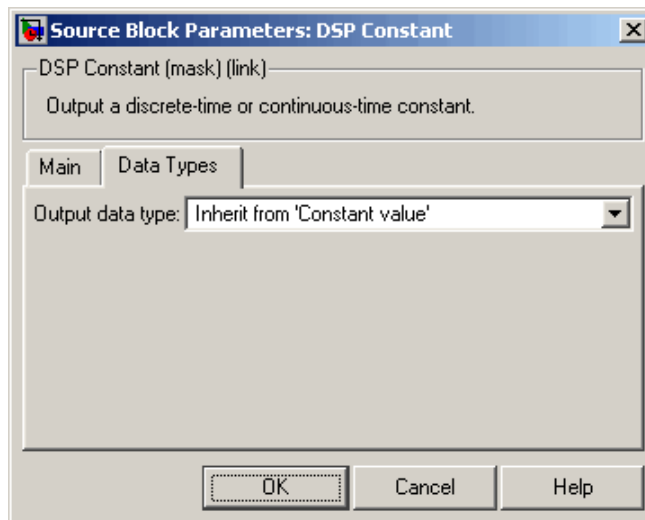
Output data type
    Specify the output data type in one of the following ways:

    • Choose one of the built-in data types from the list.

DSP Constant (Obsolete)

- Choose `Fixed-point` to specify the output data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to**, and **Fraction length** parameters.

- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `Inherit from 'Constant value'` to set the output data type and scaling to match the values of the **Constant value** parameter in the **Main** pane.

- Choose `Inherit via back propagation` to set the output data type and scaling to match the following block.

The value of this parameter overrides any data type information specified in the **Constant value** parameter in the **Main** pane, except when you select `Inherit from 'Constant value'`.

Signed
> Select to output a signed fixed-point signal. Otherwise, the signal is unsigned. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

Word length
> Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type
> Specify any built-in or fixed-point data type. You can specify fixed-point data types using the following Simulink Fixed Point functions: `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac`. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set fraction length in output to
> Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.

- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter, or when you select `User-defined` and the specified output data type is a fixed-point data type.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point

- Boolean

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

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

See Also

Constant	Simulink
Signal From Workspace	DSP System Toolbox

DWT

Purpose	Discrete wavelet transform (DWT) of input or decompose signals into subbands with smaller bandwidths and slower sample rates

Library   Transforms

dspxfrm3

Description   The DWT block is the same as the Dyadic Analysis Filter Bank block in the Multirate Filters library, but with different default settings. See the Dyadic Analysis Filter Bank block reference page for more information.
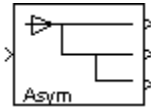
Purpose   Decompose signals into subbands with smaller bandwidths and slower sample rates or compute discrete wavelet transform (DWT)

Library   Filtering / Multirate Filters

dspmlti4

Description

Note This block always interprets input signals as frames. The frame size of the input signal must be a multiple of 2^n, where n is the value of the **Number of levels** parameter. The block decomposes the input signal into either n+1 or 2^n subbands. To decompose signals with a frame size that is not a multiple of 2^n, use the Two-Channel Analysis Subband Filter block. (You can connect multiple copies of the Two-Channel Analysis Subband Filter block to create a multilevel dyadic analysis filter bank.)

You can configure this block to compute the Discrete Wavelet Transform (DWT) or decompose a broadband signal into a collection of subbands with smaller bandwidths and slower sample rates. The block uses a series of highpass and lowpass FIR filters to repeatedly divide the input frequency range, as illustrated in the figure "n-Level Asymmetric Dyadic Analysis Filter Bank".

You can specify the filter bank's highpass and lowpass filters by providing vectors of filter coefficients. You can do so directly on the block mask, or, if you have a Wavelet Toolbox™ license, you can specify wavelet-based filters by selecting a wavelet from the **Filter** parameter. You must set the filter bank structure to asymmetric or symmetric, and specify the number of levels in the filter bank.

For the same input, the DWT configuration of this block does not produce the same results as the Wavelet Toolbox dwt function. Because DSP System Toolbox is designed for real-time implementation and Wavelet Toolbox is designed for analysis, the products handle boundary conditions and filter states differently. To make the output of the dwt

function match the DWT output of this block, complete the following steps:

1 Set the boundary condition of the dwt function to zero-padding. To do so, type dwtmode('zpd') at the MATLAB command line.

2 To match the latency of the block (implemented using FIR filters), add zeros to the input of the dwt function. The number of zeros you add must be equal to the half-length of the filter.

Input Requirements

- Input must be a vector or matrix.

- The input frame size must be a multiple of 2^n, where n is the number of filter bank levels. For example, a frame size of 16 would be appropriate for a three-level tree (16 is a multiple of 2^3).

- The block always treats input signals as frames and operates along the columns.

For an illustration of why the above input requirements exist, see the figure Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank on page 2-541.

Output Characteristics

The output characteristics vary depending on the block's parameter settings, as summarized in the following list and figure:

- **Number of levels** parameter set to n

- **Tree structure** parameter setting:

  - Asymmetric — Block produces $n+1$ output subbands

  - Symmetric — Block produces 2^n output subbands

- **Output** parameter setting can be Multiple ports or Single port. When you set the **Output** parameter to Single port, the block outputs one vector or matrix of concatenated subbands. The following figure illustrates the difference between the two settings for a

3-level asymmetric dyadic analysis filter bank. For an explanation of the illustrated output characteristics, see the table Output Characteristics for an n-Level Dyadic Analysis Filter Bank on page 2-542.

For more information about the filter bank levels and structures, see "Dyadic Analysis Filter Banks".
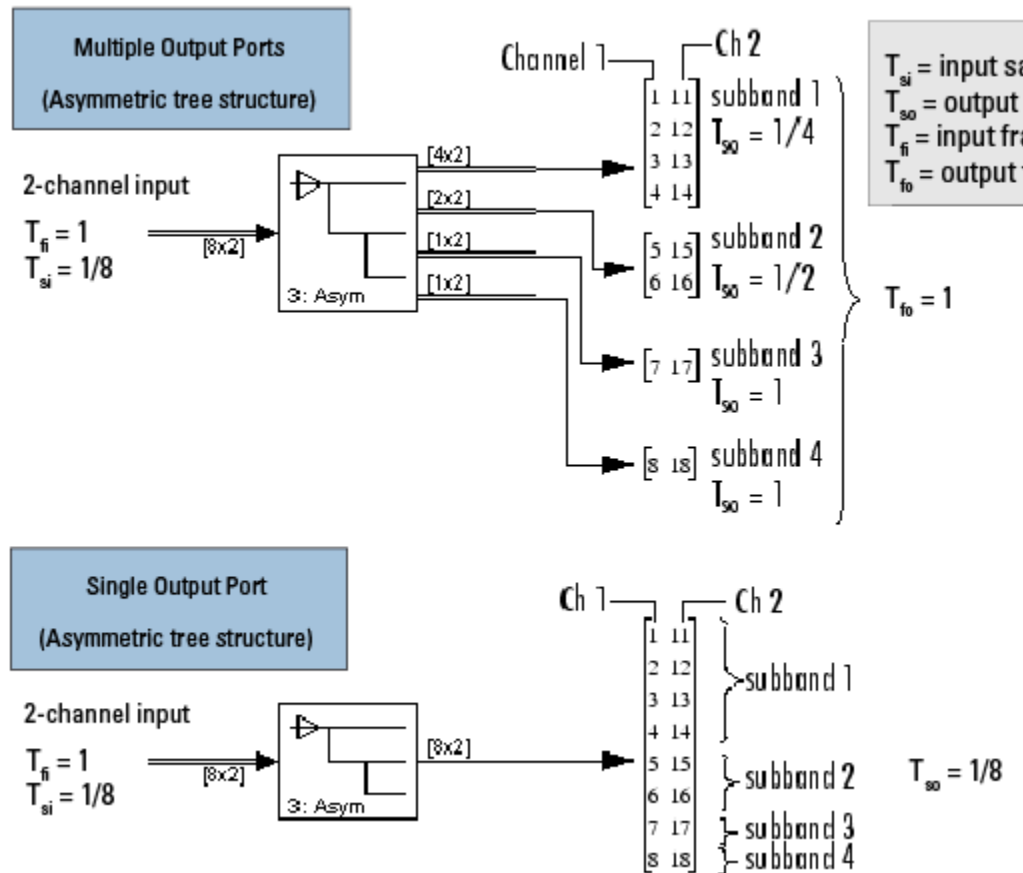
Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank

Dyadic Analysis Filter Bank

The following table summarizes the different output characteristics of the block when it is set to output from single or multiple ports.

Output Characteristics for an n-Level Dyadic Analysis Filter Bank

	Single Output Port	**Multiple Output Ports**
Output Description	Block concatenates all the subbands into one vector or matrix, and outputs the concatenated subbands from a single output port. Each output column contains subbands of the corresponding input channel.	Block outputs each subband from a separate output port. The topmost port outputs the subband with the highest frequencies. Each output column contains a subband for the corresponding input channel.
Output Frame Rate	*Not applicable*	Same as input frame rate (However, the output frame sizes can vary, so the output sample rates can vary.)
Output Dimensions (Frame Size)	Same number of rows and columns as the input.	The output has the same number of columns as the input. The number of output rows is the output frame size. For an input with frame size M_i output y_k has frame size $M_{o,k}$: • Symmetric — All outputs have the frame size, $M_i / 2^n$. • Asymmetric — The frame size of each output (except the last) is half that of the output from the previous level. The outputs from the last two output ports have the same frame size since they originate from the same level in the filter bank.

Output Characteristics for an n-Level Dyadic Analysis Filter Bank (Continued)

	Single Output Port	Multiple Output Ports
		$$M_{o,k} = \begin{cases} M_i/2^k & (1 \le k \le n) \\ M_i/2^n & (k = n+1) \end{cases}$$
Output Sample Rate	Same as input sample rate.	Though the outputs have the same frame rate as the input, they have different frame sizes than the input. Thus, the output sample rates, F_{so}, k, are different from the input sample rate, F_{si}: • Symmetric — All outputs have the sample rate $F_{si}/2^n$. • Asymmetric — $$F_{so,k} = \begin{cases} F_{si}/2^k & (1 \le k \le n) \\ F_{si}/2^n & (k = n+1) \end{cases}$$

Filter Bank Filters

You must specify the highpass and lowpass filters in the filter bank by setting the **Filter** parameter to one of the following options:

• User defined — Allows you to explicitly specify the filters with two vectors of filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters. The block uses the same lowpass and highpass filters throughout the filter bank. The two filters should be halfband filters, where each filter passes the frequency band that the other filter stops.

• Wavelet such as Biorthogonal or Daubechies — The block uses the specified wavelet to construct the lowpass and highpass filters using the Wavelet Toolbox wfilters function. Depending on the wavelet,

Dyadic Analysis Filter Bank

the block might enable either the **Wavelet order** or **Filter order
[synthesis / analysis]** parameter. (The latter parameter allows you
to specify different wavelet orders for the analysis and synthesis filter
stages.) You must have a Wavelet Toolbox license to use wavelets.

Specifying Filters with the Filter Parameter and Related Parameters

Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
User-defined	Filters based on Daubechies wavelets with wavelet order 3: • **Lowpass FIR filter coefficients =** `[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]` • **Highpass FIR filter coefficients =** `[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]`	None
Haar	None	`wfilters('haar')`
Daubechies	**Wavelet order** = 4	`wfilters('db4')`
Symlets	**Wavelet order** = 3	`wfilters('sym3')`
Coiflets	**Wavelet order** = 1	`wfilters('coif1')`
Biorthogonal	**Filter order [synthesis / analysis]** = [3/1]	`wfilters('bior3.1')`
Reverse Biorthogonal	**Filter order [synthesis / analysis]** = [3/1]	`wfilters('rbio3.1')`
Discrete Meyer	None	`wfilters('dmey')`

Examples

Wavelets

The primary application for dyadic analysis filter banks and dyadic synthesis filter banks is coding for data compression using wavelets.

At the transmitting end, the output of the dyadic analysis filter bank is fed to a lossy compression scheme, which typically assigns the number of bits for each filter bank output in proportion to the relative energy in that frequency band. This represents the more powerful signal components by a greater number of bits than the less powerful signal components.

At the receiving end, the transmission is decoded and fed to a dyadic synthesis filter bank to reconstruct the original signal. The filter coefficients of the complementary analysis and synthesis stages are designed to cancel aliasing introduced by the filtering and resampling.

See "Calculate Channel Latencies Required for Wavelet Reconstruction" for an example using the Dyadic Analysis and Dyadic Synthesis Filter Bank blocks.

Examples

See the floating-point frame-based version of the DSP System Toolbox Wavelet Reconstruction and Noise Reduction example, which uses the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks.

Dialog Box

The parameters displayed in the block dialog vary depending on the setting of the **Filter** parameter. Only some of the parameters described below are visible in the dialog box at any one time.

Dyadic Analysis Filter Bank

Function Block Parameters: Dyadic Analysis Filter Bank

Dyadic Analysis Filter Bank (mask) (link)

Compute the Discrete Wavelet Transform or decompose a signal into subbands with smaller bandwidths and slower sample rates. This block uses a filter bank with lowpass and highpass FIR filters that you specify either directly on the block mask, or using wavelets from the Wavelet Toolbox. The lowpass and highpass filters are usually half-band filters designed to complement each other.

Inputs are always interpreted as frames. The frame size must be a multiple of 2^n, where you specify n in the 'Number of levels' parameter. When the 'Output' parameter is set to 'Multiple ports', the block outputs each subband from a different port as a vector or matrix. The topmost port outputs the subband with the highest frequency band. When the 'Output' parameter is set to 'Single port', the block outputs one vector or matrix of concatenated subbands.

Parameters

Filter: User-defined

Lowpass FIR filter coefficients:

[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]

Highpass FIR filter coefficients:

[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]

Number of levels:

2

Tree structure: Asymmetric

Output: Multiple ports

OK   Cancel   Help   Apply

Filter

The type of filter used to determine the high- and low-pass FIR filters in the filter bank:

Select `User defined` to explicitly specify the filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

Select a wavelet such as `Biorthogonal` or `Daubechies` to specify a wavelet-based filter. The block uses the Wavelet Toolbox `wfilters` function to construct the filters. Extra parameters such as **Wavelet order** or **Filter order [synthesis / analysis]** might become enabled. For a list of the supported wavelets, see Specifying Filters with the Filter Parameter and Related Parameters on page 2-544.

Lowpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) that specifies coefficients used by all the lowpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a Daubechies wavelet with wavelet order 3.

Highpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) that specifies coefficients used by all the highpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a Daubechies wavelet with wavelet order 3.

Dyadic Analysis Filter Bank

Wavelet order

The order of the wavelet selected in the **Filter** parameter. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in the Specifying Filters with the Filter Parameter and Related Parameters on page 2-544 table.

Filter order [synthesis / analysis]

The order of the wavelet for the synthesis and analysis filter stages. For example, when you set the **Filter** parameter to `Biorthogonal` and set the **Filter order [synthesis / analysis]** parameter to `[2 / 6]`, the block calls the `wfilters` function with input argument `'bior2.6'`. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in Specifying Filters with the Filter Parameter and Related Parameters on page 2-544.

Number of levels

The number of filter bank levels. An n-level asymmetric structure has $n+1$ outputs, and an n-level symmetric structure has 2^n outputs, as shown in the figures "n-Level Asymmetric Dyadic Analysis Filter Bank" and "n-Level Symmetric Dyadic Analysis Filter Bank". The block's icon changes depending on the value of this parameter.

The default setting of this parameter is `2`.

Tree structure

The structure of the filter bank: `Asymmetric`, or `Symmetric`. See the figures "n-Level Asymmetric Dyadic Analysis Filter Bank" and "n-Level Symmetric Dyadic Analysis Filter Bank".

The default setting of this parameter is `Asymmetric` for the Dyadic Analysis Filter Bank block, and `Symmetric` for the DWT block.

Output

Set to `Multiple ports` to output each output subband on a separate port (the topmost port outputs the subband with the highest frequency band). Set to `Single port` to concatenate the subbands into one vector or matrix and output the concatenated

subbands on a single port. For more information, see "Output Characteristics" on page 2-540.

The default setting of this parameter is `Multiple ports` for the Dyadic Analysis Filter Bank block, and `Single port` for the DWT block.

References Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

See Also

DWT	DSP System Toolbox
Dyadic Synthesis Filter Bank	DSP System Toolbox
Two-Channel Analysis Subband Filter	DSP System Toolbox

Dyadic Synthesis Filter Bank

Purpose   Reconstruct signals from subbands with smaller bandwidths and slower sample rates or compute inverse discrete wavelet transform (IDWT)

Library   Filtering / Multirate Filters

`dspmlti4`

Description
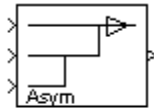
Note This block always does frame-based processing, and its inputs must be of certain sizes. To use input subbands that do not fit the criteria of this block, use the Two-Channel Synthesis Subband Filter block. (You can connect multiple copies of the Two-Channel Synthesis Subband Filter block to create a multilevel dyadic synthesis filter bank.)

You can configure this block to compute the inverse discrete wavelet transform (IDWT) or reconstruct a signal from subbands with smaller bandwidths and slower sample rates. When the block computes the inverse discrete wavelet transform (IDWT) of the input, the output has the same dimensions as the input. Each column of the output is the IDWT of the corresponding input column. When reconstructing a signal, the block uses a series of highpass and lowpass FIR filters to reconstruct the signal from the input subbands, as illustrated in the figure "n-Level Asymmetric Dyadic Synthesis Filter Bank". The reconstructed signal has a wider bandwidth and faster sample rate than the input subbands.

You can specify the filter bank's highpass and lowpass filters by providing vectors of filter coefficients. You can do so directly on the block mask, or, if you have a Wavelet Toolbox license, you can specify wavelet-based filters by selecting a wavelet from the **Filter** parameter. You must set the filter bank structure to asymmetric or symmetric, and specify the number of levels in the filter bank.

When you set the **Input** parameter to `Multiple ports`, you must provide each subband to the block through a different input port as a vector or matrix. You should input the highest frequency band through the topmost port. When you set the **Input** parameter to `Single port`, the block input must be a vector or matrix of concatenated subbands.

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

This block automatically computes wavelet-based perfect reconstruction filters when the wavelet selection in the **Filter** parameter of this block is the *same* as the **Filter** parameter setting of the corresponding Dyadic Analysis Filter Bank block. The use of wavelets requires a Wavelet Toolbox license. To learn how to design your own perfect reconstruction filters, see "References" on page 2-560.

Input Requirements

The inputs to this block are usually the outputs of a Dyadic Analysis Filter Bank block. Since the Dyadic Analysis Filter Bank block can output from either a single port or multiple ports, the Dyadic Synthesis Filter Bank block accepts inputs to either a single port or multiple ports.

The **Input** parameter sets whether the block accepts inputs from a single port or multiple ports, and thus determines the input requirements, as summarized in the following lists and figure.

Note Any output of a Dyadic Analysis Filter Bank block whose parameter settings match the corresponding settings of this block is a valid input to this block. For example, the setting of the Dyadic Analysis Filter Bank block parameter, **Output**, must be the same as this block's **Input** parameter (`Single port` or `Multiple ports`).

Valid Inputs for Input Set to Single Port

- Inputs must be vectors or matrices of concatenated subbands. The block always interprets the inputs as sample based.

- Each input column contains the subbands for an independent signal.

- Upper input rows contain the high-frequency subbands, and the lower rows contain the low-frequency subbands.

Valid Inputs for Input Set to Multiple Ports

- Each subband must be provided as a vector or matrix to separate block input ports. The block always interprets the inputs as frame based.

- The columns of each input contains a subband for an independent signal.

- The input to the topmost input port is the subband containing the highest frequencies, and the input to the bottommost port is the subband containing the lowest frequencies.
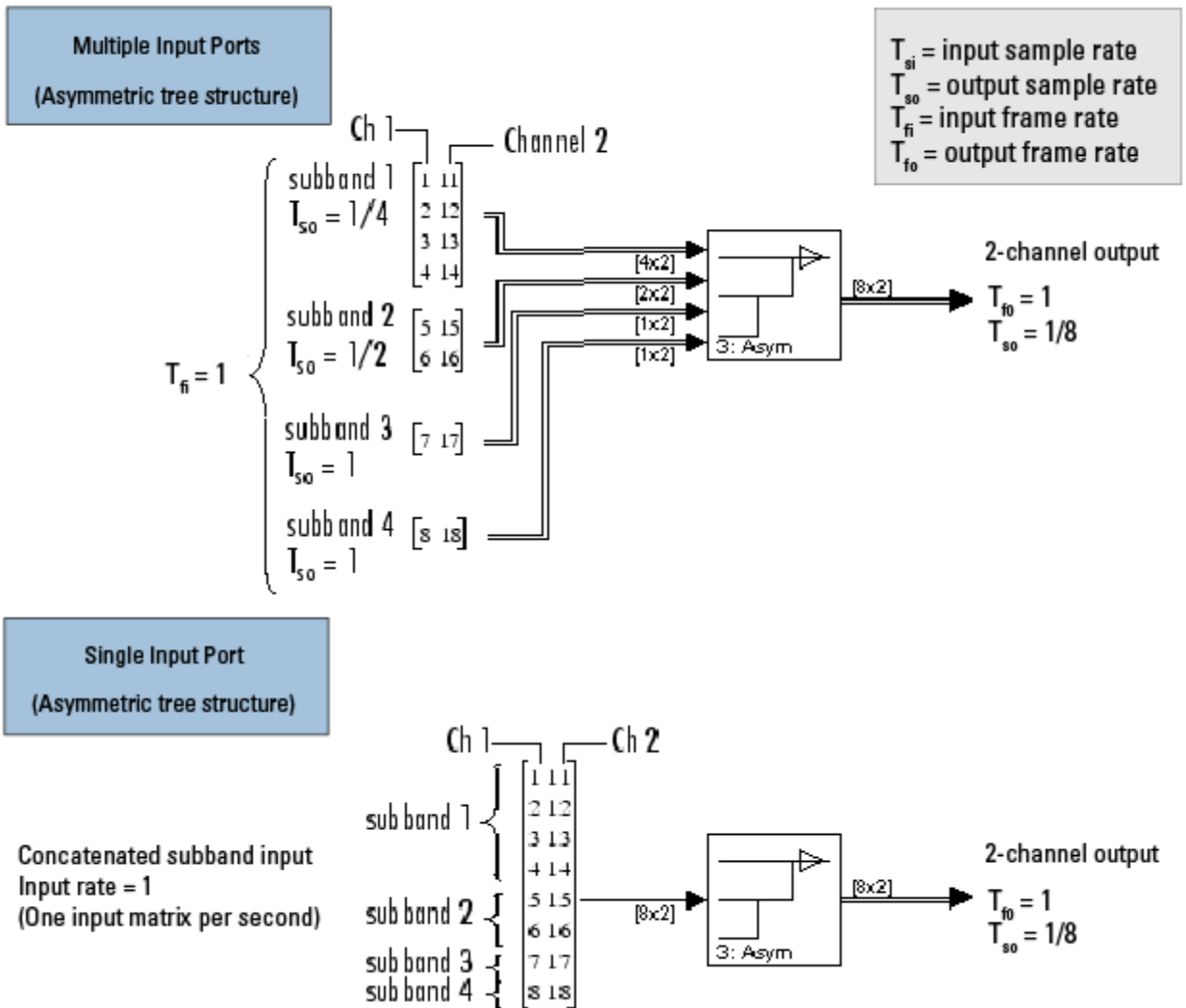
Valid Inputs to a 3-Level Asymmetric Dyadic Synthesis Filter Bank

Dyadic Synthesis Filter Bank

For general information about the filter banks, see "Dyadic Synthesis Filter Banks".

Output Characteristics

The following table summarizes the output characteristics for both types of inputs. For an illustration of why the output characteristics exist, see the figure Valid Inputs to a 3-Level Asymmetric Dyadic Synthesis Filter Bank on page 2-553.

	Input = Multiple ports	Input = Single port (Concatenated Subband Inputs)
Output Frame Rate	Same as the input frame rate.	Same as the input rate (the rate of the concatenated subband inputs).
Output Frame Dimensions	• The output has the same number of columns as the inputs. • The number of output rows depends on the tree structure of the filter bank: ▪ **Asymmetric** — The number of output rows is twice the number of rows in the input to the topmost input port. ▪ **Symmetric** — The number of output rows is the product of the number of input ports and the number of rows in an input to any input port.	The output has the same number of rows and columns as the input.

For general information about the filter banks, see "Dyadic Synthesis Filter Banks".

Filter Bank Filters

You must specify the highpass and lowpass filters in the filter bank by setting the **Filter** parameter to one of the following options:

- `User defined` — Allows you to explicitly specify the filters with two vectors of filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters. The block uses the same lowpass and highpass filters throughout the filter bank. The two filters should be halfband filters, where each filter passes the frequency band that the other filter stops. To use this block to perfectly reconstruct a signal decomposed by a Dyadic Analysis Filter Bank block, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. To learn how to design your own perfect reconstruction filters, see "References" on page 2-560.

- Wavelet such as `Biorthogonal` or `Daubechies` — The block uses the specified wavelet to construct the lowpass and highpass filters using the Wavelet Toolbox function `wfilters`. Depending on the wavelet, the block might enable either the **Wavelet order** or **Filter order [synthesis / analysis]** parameter. (The latter parameter allows you to specify different wavelet orders for the analysis and synthesis filter stages.) To use this block to reconstruct a signal decomposed by a Dyadic Analysis Filter Bank block, you must set both blocks to use the same wavelets with the same order. You must have a Wavelet Toolbox license to use wavelets.

Dyadic Synthesis Filter Bank

Specifying Filters with the Filter Parameter and Related Parameters

Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
User-defined	Filters based on Daubechies wavelets with wavelet order 3: • **Lowpass FIR filter coefficients =** `[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]` • **Highpass FIR filter coefficients =** `[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]`	None
Haar	None	`wfilters('haar')`
Daubechies	**Wavelet order =** 4	`wfilters('db4')`
Symlets	**Wavelet order =** 3	`wfilters('sym3')`
Coiflets	**Wavelet order =** 1	`wfilters('coif1')`
Biorthogonal	**Filter order [synthesis / analysis] =** [3/1]	`wfilters('bior3.1')`
Reverse Biorthogonal	**Filter order [synthesis / analysis] =** [3/1]	`wfilters('rbio3.1')`
Discrete Meyer	None	`wfilters('dmey')`

Examples   See "Examples" on page 2-545 on the Dyadic Analysis Filter Bank block reference page.

Dialog Box   The parameters displayed in the block dialog vary depending on the setting of the **Filter** parameter. Only some of the parameters described below are visible in the dialog box at any one time.

Function Block Parameters: Dyadic Synthesis Filter Bank

Dyadic Synthesis Filter Bank (mask) (link)

Compute the Inverse Discrete Wavelet Transform or reconstruct a signal from subbands with smaller bandwidths and slower sample rates. This block uses a filter bank with lowpass and highpass FIR filters that you specify either directly on the block mask, or using wavelets from the Wavelet Toolbox. The lowpass and highpass filters are usually half-band filters designed to complement each other.

When 'Input' is set to 'Multiple ports', you must provide each subband to the block through a different input port as a vector or matrix. You should input the highest frequency band through the topmost port. When 'Input' is set to 'Single port', the block input must be a vector or matrix of concatenated subbands.

Parameters

Filter: User-defined

Lowpass FIR filter coefficients:

[0.3327 0.8069 0.4599 -0.1350 -0.0854 0.0352]

Highpass FIR filter coefficients:

[0.0352 0.0854 -0.1350 -0.4599 0.8069 -0.3327]

Number of levels:

2

Tree Structure: Asymmetric

Input: Multiple ports

OK   Cancel   Help   Apply

Dyadic Synthesis Filter Bank

Filter

The type of filter used to determine the high- and low-pass FIR filters in the filter bank:

- Select `User defined` to explicitly specify the filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

- Select a wavelet such as `Biorthogonal` or `Daubechies` to specify a wavelet-based filter. The block uses the Wavelet Toolbox `wfilters` function to construct the filters. Extra parameters such as **Wavelet order** or **Filter order [synthesis / analysis]** might become enabled. For a list of the supported wavelets, see the table Specifying Filters with the Filter Parameter and Related Parameters on page 2-556.

Lowpass FIR filter coefficients

A vector of filter coefficients (descending powers of *z*) that specifies coefficients used by all the lowpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. To perfectly reconstruct a signal decomposed by the Dyadic Analysis Filter Bank, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is not perfect. The default values of this parameter specify a perfect reconstruction filter for the default settings of the Dyadic Analysis Filter Bank (based on a Daubechies wavelet with wavelet order 3).

Highpass FIR filter coefficients

A vector of filter coefficients (descending powers of *z*) that specifies coefficients used by all the highpass filters in the filter bank. This parameter is enabled when you set **Filter** to User defined. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. To perfectly reconstruct a signal decomposed by the Dyadic Analysis Filter Bank, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is not perfect. The default values of this parameter specify a perfect reconstruction filter for the default settings of the Dyadic Analysis Filter Bank (based on a Daubechies wavelet with wavelet order 3).

Wavelet order

The order of the wavelet selected in the **Filter** parameter. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in the table Specifying Filters with the Filter Parameter and Related Parameters on page 2-556.

Filter order [synthesis / analysis]

The order of the wavelet for the synthesis and analysis filter stages. For example, when you set the **Filter** parameter to **Biorthogonal** and set the **Filter order [synthesis / analysis]** parameter to [2 / 6], the block calls the wfilters function with input argument 'bior2.6'. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in Specifying Filters with the Filter Parameter and Related Parameters on page 2-556.

Number of levels

The number of filter bank levels. An *n*-level asymmetric structure has $n+1$ inputs, and an *n*-level symmetric structure has 2^n inputs, as shown in "n-Level Asymmetric Dyadic Synthesis Filter Bank" and "n-Level Symmetric Dyadic Synthesis Filter Bank".

The default setting of this parameter is 2.

Dyadic Synthesis Filter Bank

Tree structure

The structure of the filter bank: `Asymmetric`, or `Symmetric`. See the figures "n-Level Asymmetric Dyadic Synthesis Filter Bank" and "n-Level Symmetric Dyadic Synthesis Filter Bank".

The default setting of this parameter is `Asymmetric` for the Dyadic Synthesis Filter Bank block, and `Symmetric` for the IDWT block.

Input

Set to `Multiple ports` to accept each input subband at a separate port (the topmost port accepts the subband with the highest frequency band). Set to `Single port` to accept one vector or matrix of concatenated subbands at a single port. For more information, see "Input Requirements" on page 2-551.

The default setting of this parameter is `Multiple ports` for the Dyadic Synthesis Filter Bank block, and `Single port` for the IDWT block.

References   Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets.* West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks.* Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks.* Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

See Also

Dyadic Analysis Filter Bank	DSP System Toolbox
IDWT	DSP System Toolbox
Two-Channel Synthesis Subband Filter	DSP System Toolbox

See "Multirate and Multistage Filters" for related information.

Edge Detector

Purpose   Detect transition from zero to nonzero value

Library   Signal Management / Switches and Counters

dspswit3
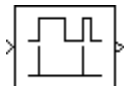
Description The Edge Detector block generates an impulse (the value 1) in a given output channel when the corresponding channel of the input transitions from zero to a nonzero value. When the input does not transition from zero to a nonzero value, the block generates a zero in the corresponding output channel.

The output has the same dimension and sample rate as the input. When you set the **Input processing** parameter to Columns as channels (frame based), the block counts an edge that is split across two consecutive frames in the frame that contains the nonzero value. For example, if there is a zero at the bottom of the first frame and a nonzero value at the top of the second frame, the block counts the edge in the second frame.

Examples   In the ex_edgedetector_ref model, the **Input processing** parameter of the Edge Detector block is set to Columns as channels (frame based). Thus, the block interprets the 3-by-2 input as a multichannel signal with a frame size of 3. The Matrix Concatenate block concatenates the two input channels of the original signal with the two output channels of the Edge Detector block to create the four-channel workspace variable sp_examples_yout.
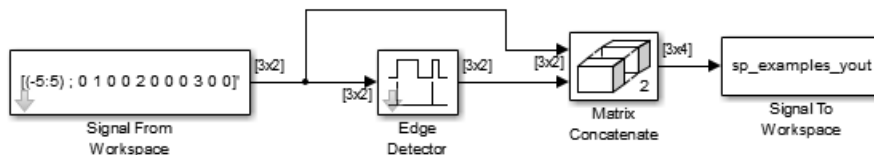
As shown in the following figure, the block finds edges at sample 7 in channel 1, and at samples 2, 5, and 9 in channel 2.
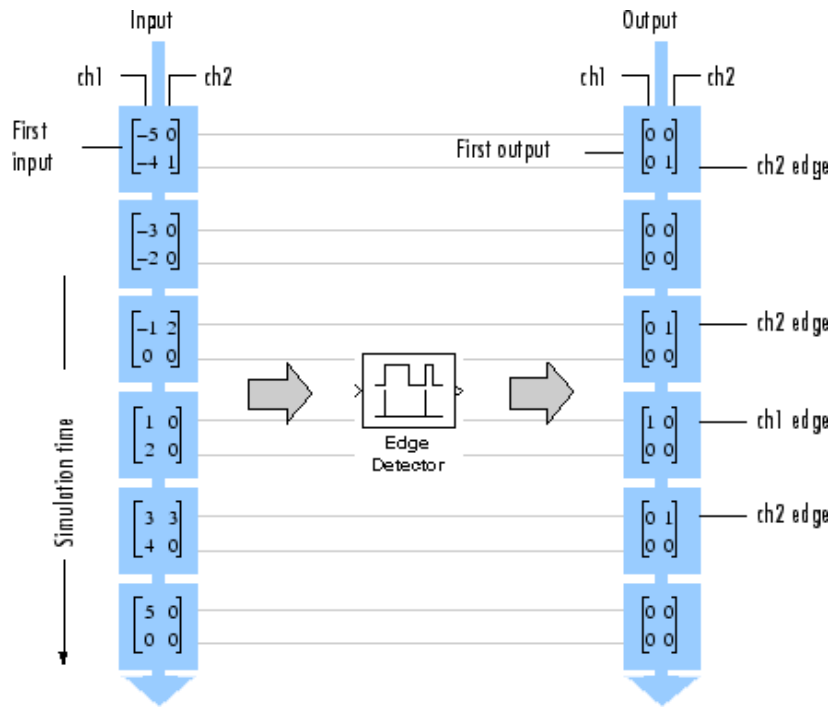
Edge Detector

Dialog Box

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)

- Boolean — The block might output Boolean values depending on the input data type, and whether Boolean support is enabled or disabled.

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

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

- Enumerated

See Also

Counter	DSP System Toolbox
Event-Count Comparator	DSP System Toolbox

Event-Count Comparator

Purpose   Detect threshold crossing of accumulated nonzero inputs

Library   Signal Management / Switches and Counters

dspswit3

Description   The Event-Count Comparator block records the number of nonzero
inputs to the Data port during the period that the block is enabled by a
high signal (the value 1) at the Int port. Both inputs must be scalars.

When the number of accumulated nonzero inputs first equals the **Event
threshold** setting, the block waits one additional sample interval, and
then sets the output high (1). The block holds the output high until
recording is restarted by a low-to-high (0-to-1) transition at the Int port.

The Event-Count Comparator block accepts real and complex
floating-point and fixed-point inputs. However, because the block
has discrete state, it does not support constant or continuous sample
times. Therefore, at least one input or output port of the Event-Count
Comparator block must be connected to a block whose **Sample time**
parameter is discrete. The Event-Count Comparator block inherits this
non-infinite discrete sample time.

Examples   In the ex_eventcountcomp_ref model, the Event-Count Comparator
block (**Event threshold** = 3) detects two threshold crossings in the
input to the Data port, one at sample 4 and one at sample 12.
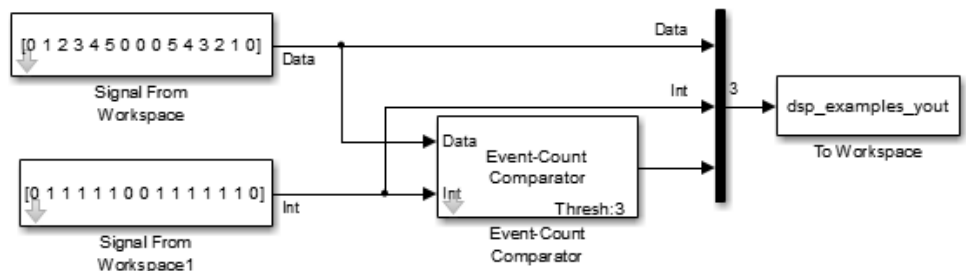
All inputs and outputs are multiplexed into the workspace variable
yout, whose contents are shown in the figure below. The two left
columns in the illustration show the inputs to the Data and Int ports,
the center column shows the state of the block's internal counter, and
the right column shows the block's output.

Data	Int	Nonzero count	Output
[0]	[0]	[0]	[0]
[1]	[1] start count	[1]	[0]
[2]	[1]	[2]	[0]
[3]	[1]	[3] equals threshold	[0]
[4]	[1]	[4] exceeds threshold	[1]
[5]	[1]	[5]	[1]
[0]	[0] stop count	[5]	[1]
[0]	[0]	[5]	[1]
[0]	[1] restart count,	[0]	[0]
[5]	[1]	[1]	[0]
[4]	[1]	[2]	[0]
[3]	[1]	[3]	[0]
[2]	[1]	[4] equals threshold	[1]
[1]	[1]	[5] exceeds threshold	[1]
[0]	[0]	[5]	[1]
	stop count		
⋮	⋮	⋮	⋮
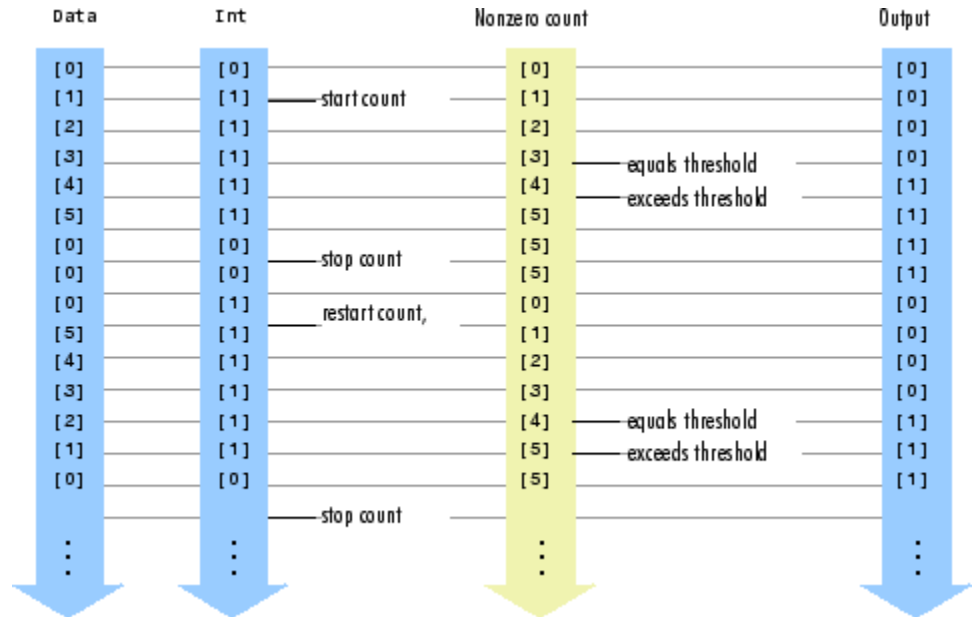
Event-Count Comparator
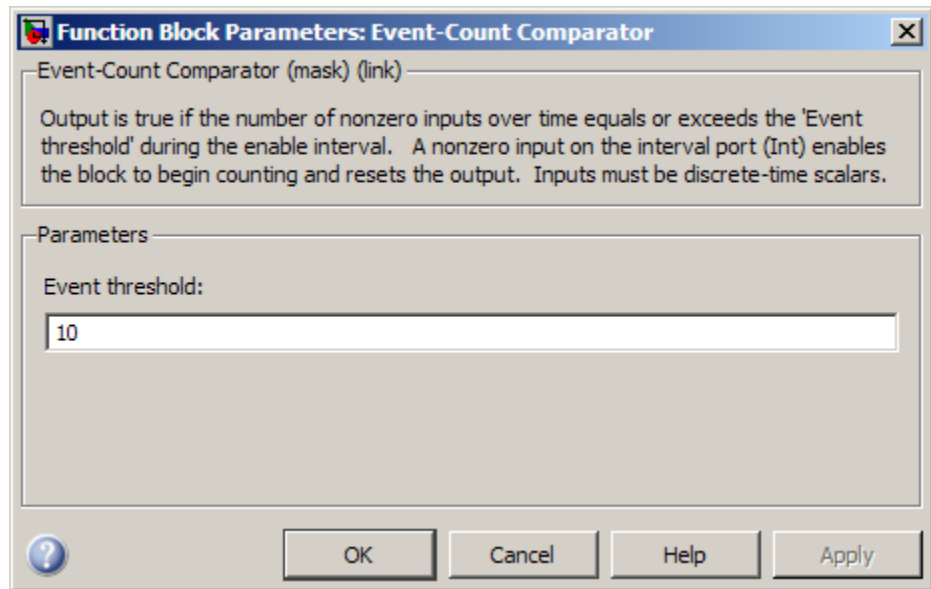
**Dialog
Box**

Event threshold
> Specify the value against which to compare the number of nonzero inputs. Tunable.

**Supported
Data
Types**

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers
- Enumerated

See Also

Counter	DSP System Toolbox
Edge Detector	DSP System Toolbox

Extract Diagonal

Purpose   Extract main diagonal of input matrix

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3
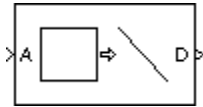
Description   The Extract Diagonal block populates the unoriented output vector with the elements on the main diagonal of the *M*-by-*N* input matrix A.

```
D = diag(A)      Equivalent MATLAB code
```

The output vector has length min(M,N).
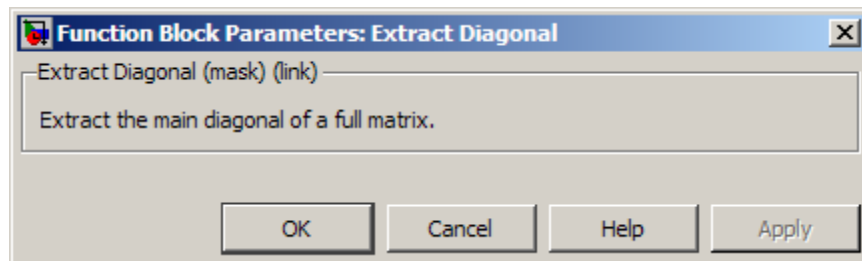
Dialog Box

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Boolean — Block outputs are always Boolean.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

Constant Diagonal Matrix	DSP System Toolbox
Create Diagonal Matrix	DSP System Toolbox

Extract Triangular Matrix	DSP System Toolbox
diag	MATLAB

Extract Triangular Matrix

Purpose   Extract lower or upper triangle from input matrices

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3
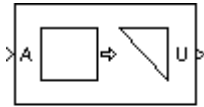
Description   The Extract Triangular Matrix block creates a triangular matrix output
from the upper or lower triangular elements of an *M*-by-*N* input matrix.
The block treats length-*M* unoriented vector inputs as an *M*-by-1 matrix.

The **Extract** parameter selects between the two components of the
input:
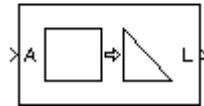
- Upper — Copies the elements on and above the main diagonal of the
  input matrix to an output matrix of the same size. The first *row* of
  the output matrix is therefore identical to the first *row* of the input
  matrix. The elements below the main diagonal of the output matrix
  are zero.

- 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.

Examples   The ex_extracttriang_ref model below shows the extraction of upper
and lower triangles from a 5-by-3 input matrix.
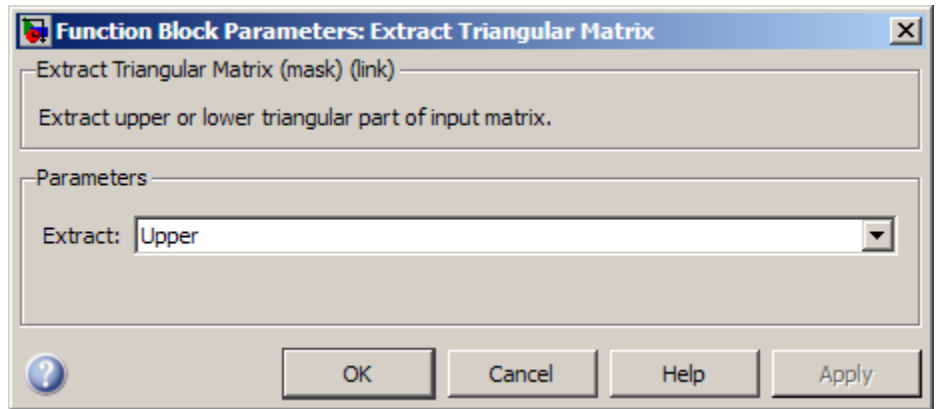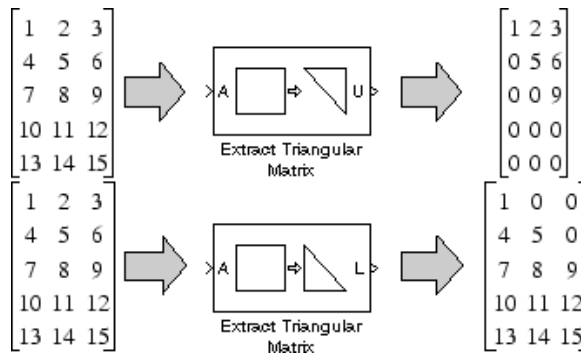
Dialog Box

Extract

The component of the matrix to copy to the output: upper triangle or lower triangle.

Extract Triangular Matrix

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
U	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
L	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Autocorrelation LPC	DSP System Toolbox
Cholesky Factorization	DSP System Toolbox
Constant Diagonal Matrix	DSP System Toolbox
Extract Diagonal	DSP System Toolbox

Forward Substitution	DSP System Toolbox
LDL Factorization	DSP System Toolbox
LU Factorization	DSP System Toolbox
`tril`	MATLAB
`triu`	MATLAB

Fast Block LMS Filter

Purpose   Compute output, error, and weights using LMS adaptive algorithm

Library   Filtering / Adaptive Filters

dspadpt3

Description

```
  Input                    Output
  Desired   Fast Block
  Step-size   LMS          Error
  Adapt
                           Wts
  Reset
      Fast Block LMS Filter
```
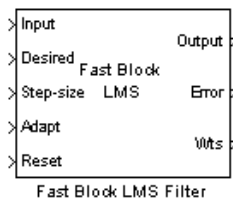
The Fast Block LMS Filter block implements an adaptive least mean-square (LMS) filter, where the adaptation of the filter weights occurs once for every block of data samples. The block estimates the filter weights, or coefficients, needed to convert the input signal into the desired signal. Connect the signal you want to filter to the Input port. The input signal can be a scalar or a column vector. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal. The Error port outputs the result of subtracting the output signal from the desired signal.

The block calculates the filter weights using the Block LMS Filter equations. For more information, see Block LMS Filter. The Fast Block LMS Filter block implements the convolution operation involved in the calculations of the filtered output, y, and the weight update function in the frequency domain using the FFT algorithm used in the Overlap-Save FFT Filter block. See Overlap-Save FFT Filter for more information.

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

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

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

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

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

Enter the initial filter weights, $\mathbf{w}(0)$, as a vector or a scalar in the **Initial value of filter weights** text box. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

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

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

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

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

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

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

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

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

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

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

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

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

Fast Block LMS Filter
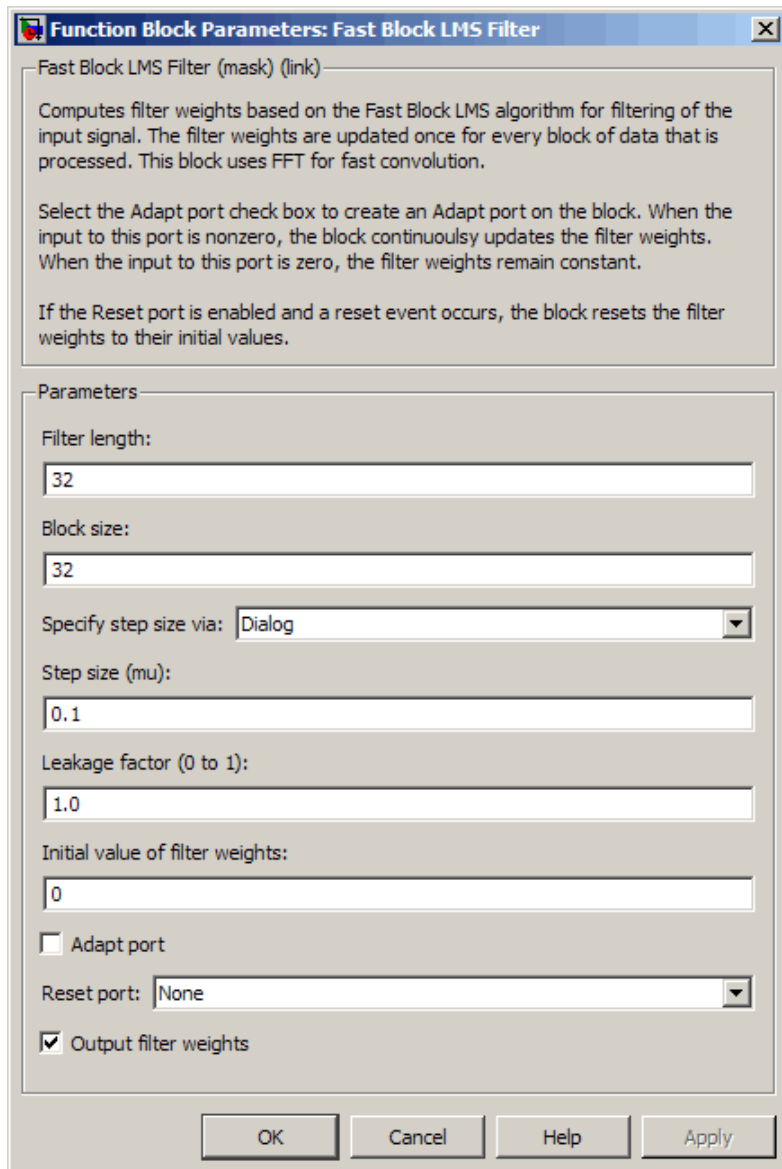
Dialog Box

Filter length

Enter the length of the FIR filter weights vector. The sum of the **Block size** and the **Filter length** must be a power of 2.

Block size

Enter the number of samples to acquire before the filter weights are updated. The number of rows in the input must be an integer multiple of the **Block size**. The sum of the **Block size** and the **Filter length** must be a power of 2.

Specify step-size via

Select `Dialog` to enter a value for mu, or select `Input port` to specify mu using the Step-size input port.

Step-size (mu)

Enter the step-size. Tunable.

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \le 1$. Tunable.

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Adapt port

Select this check box to enable the Adapt input port.

Reset input

Select this check box to enable the Reset input port.

Output filter weights

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

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

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Desired	• Must be the same as Input
Step-size	• Must be the same as Input

Fast Block LMS Filter

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

See Also

Block LMS Filter	DSP System Toolbox
Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
LMS Filter	DSP System Toolbox
RLS Filter	DSP System Toolbox

See "Adaptive Filters in Simulink" for related information.

Purpose   Fast Fourier transform (FFT) of input

Library   Transforms

                dspxfrm3

Description   The FFT block computes the fast Fourier transform (FFT) of each row
                of a sample-based 1-by-P input vector, u, or across the first dimension
                (P) of an N-D input array, u. For user-specified FFT lengths, not equal
                to P, zero padding or modulo-length data wrapping occurs before the
                FFT operation, as per Orfanidis [1]:

```
y = fft(u,M)  % P   M
```

```
y(:,l) = fft(datawrap(u(:,l),M))  % P > M; l = 1,...,N
```

When the input length, P, is greater than the FFT length, M, you
may see magnitude increases in your FFT output. These magnitude
increases occur because the FFT block uses modulo-M data wrapping to
preserve all available input samples.

To avoid such magnitude increases, you can truncate the length of your
input sample, P, to the FFT length, M. To do so, place a Pad block before
the FFT block in your model.

The kth entry of the lth output channel, $y(k, l)$, equals the kth point of
the M-point discrete Fourier transform (DFT) of the lth input channel:

$$y(k,l) = \sum_{p=1}^{P} u(p,l)e^{-j2\pi(p-1)(k-1)/M} \qquad k = 1,\dots,M$$

The block uses one of two possible FFT implementations. You can select an implementation based on the FFTW library [3], [4], or an implementation based on a collection of Radix-2 algorithms. You can select Auto to allow the block to choose the implementation.

FFTW Implementation

The FFTW implementation provides an optimized FFT calculation including support for power-of-two and non-power-of-two transform lengths in both simulation and code generation. Generated code using the FFTW implementation will be restricted to those computers which are capable of running MATLAB. The input data type must be floating-point.

Radix-2 Implementation

The Radix-2 implementation supports bit-reversed processing, fixed or floating-point data, and allows the block to provide portable C-code generation using the Simulink Coder. The dimension *M* of the *M*-by-*N* input matrix, must be a power of two. To work with other input sizes, use the Pad block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation.

With Radix-2 selected, the block implements one or more of the following algorithms:

- Butterfly operation
- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm

Radix-2 Algorithms for Real and Complex Signals

Complexity of Input	Output Ordering	Algorithms Used for FFT Computation
Complex	Linear	Bit-reversed operation and radix-2 DIT
Complex	Bit-reversed	Radix-2 DIF
Real	Linear	Bit-reversed operation and radix-2 DIT in conjunction with the half-length and double-signal algorithms
Real	Bit-reversed	Radix-2 DIF in conjunction with the half-length and double-signal algorithms

The efficiency of the FFT algorithm can be enhanced for real input signals by forming complex-valued sequences from the real-valued sequences prior to the computation of the DFT. When there are $2N+1$ real input channels, the FFT block forms these complex-valued sequences by applying the double-signal algorithm to the first $2N$ input channels, and the half-length algorithm to the last odd-numbered channel.

For real input signals with fixed-point data types, it is possible to see different numerical results in the output of the last odd numbered channel, even when all input channels are identical. This numerical difference results from differences in the double-signal algorithm and the half-length algorithm.

You can eliminate this numerical difference in two ways:

- Using full precision arithmetic for fixed-point input signals
- Changing the input data type to floating point

For more information on the double-signal and half-length algorithms, see Proakis [2]. "Efficient Computation of the DFT of Two Real Sequences" on page 475 describes the double signal algorithm. "Efficient Computation of the DFT of a 2N-Point Real Sequence" on page 476 describes the half-length algorithm.

Radix-2 Optimization for the Table of Trigonometric Values

In certain situations, the block's Radix–2 algorithm computes all the possible trigonometric values of the twiddle factor

$$e^{j\frac{2\pi k}{K}}$$

where K is the greater value of either M or N and $k = 0, \cdots, K - 1$. The block stores these values in a table and retrieves them during simulation. The number of table entries for fixed-point and floating-point is summarized in the following table:

Number of Table Entries for N-Point FFT	
floating-point	3 N/4
fixed-point	N

Fixed-Point Data Types

The following diagrams show the data types used in the FFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the FFT dialog box as discussed in "Dialog Box" on page 2-589.

Inputs to the FFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. The block multiplies in a twiddle factor before each butterfly stage in a decimation-in-time FFT and after each butterfly stage in a decimation-in-frequency FFT.

Decimation-in-time IFFT
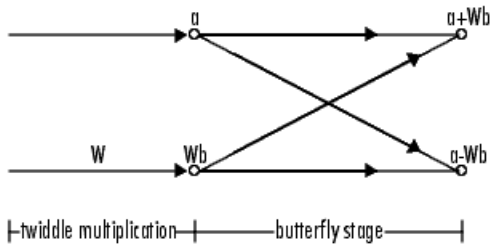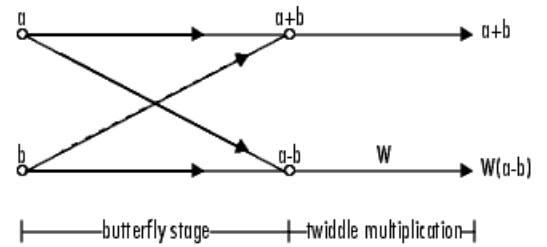
|—twiddle multiplication—|————butterfly stage————|

Decimation-in-frequency IFFT

|————butterfly stage————|—twiddle multiplication—|

Butterfly stage data types
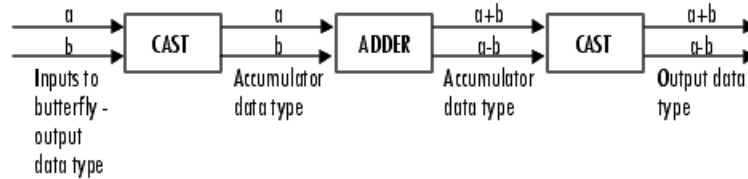
Twiddle multiplication data types

The output of the multiplier appears in the accumulator data type because both of the inputs to the multiplier are complex. For details on the complex multiplication performed, see "Multiplication Data Types".

Note When the block input is fixed point, all internal data types are signed fixed point.

Dialog Box

The **Main** pane of the FFT block dialog appears as follows.

Function Block Parameters: FFT

FFT

Compute the fast Fourier transform (FFT) across the first dimension of the input.

When you set the 'FFT implementation' parameter to 'Radix-2', the FFT length must b

| Main | Data Types |

Parameters

FFT implementation: Auto

☐ Output in bit-reversed order

☐ Scale result by FFT length

☑ Inherit FFT length from input dimensions

OK   Ca
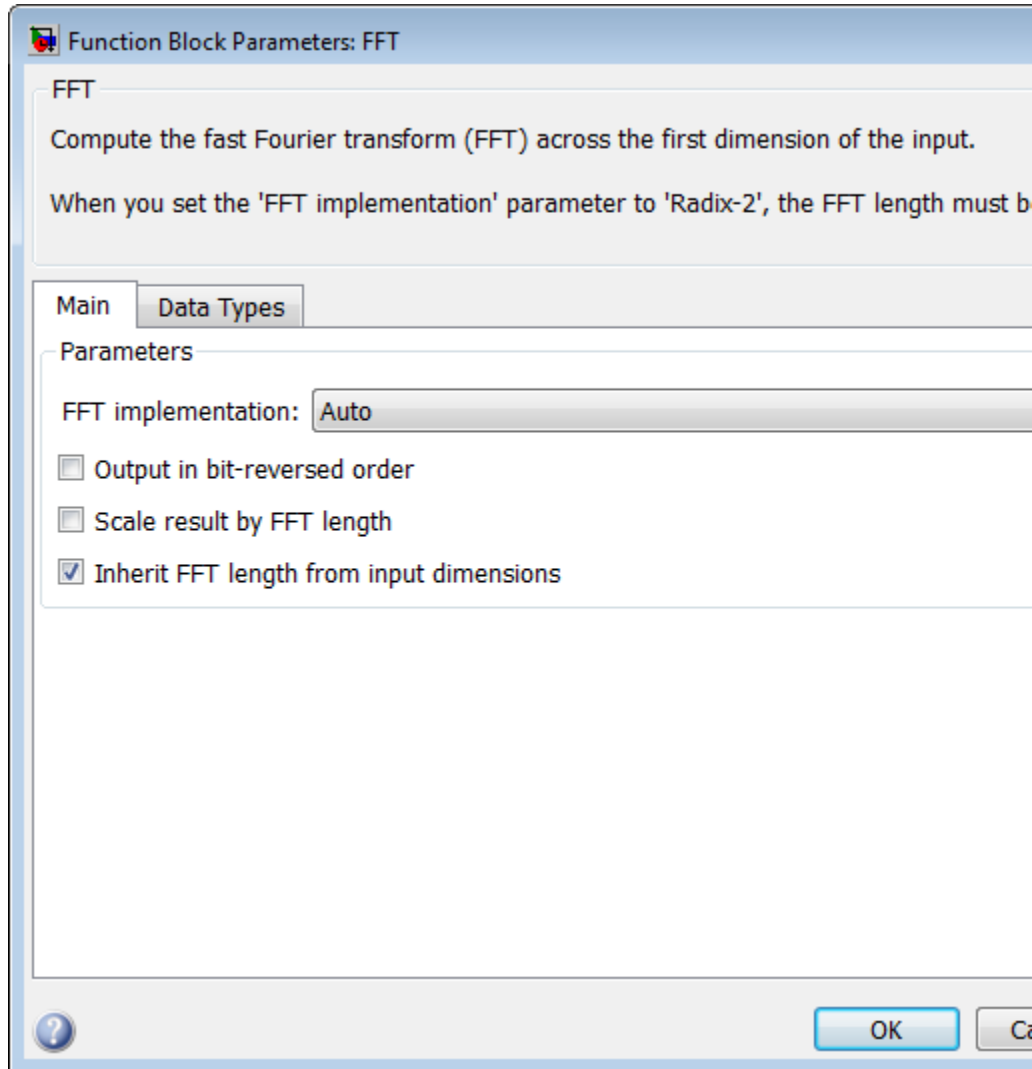
FFT implementation

Set this parameter to FFTW [3], [4] to support an arbitrary length input signal. The block restricts generated code with FFTW implementation to host computers capable of running MATLAB.

Set this parameter to Radix-2 for bit-reversed processing, fixed or floating-point data, or for portable C-code generation using the Simulink Coder. The dimension *M* of the *M*-by-*N* input matrix, must be a power of two. To work with other input sizes, use the Pad block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation. See "Radix-2 Implementation" on page 2-584.

Set this parameter to Auto to let the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

Output in bit-reversed order

Designate the order of the output channel elements relative to the ordering of the input elements. When you select this check box, the output channel elements appear in bit-reversed order relative to the input ordering. If you clear this check box, the output channel elements appear in linear order relative to the input ordering.

Linearly ordering the output requires extra data sorting manipulation, so in some situations it might be better to output in bit-reversed order.

Note The FFT block calculates its output in bit-reversed order. Linearly ordering the FFT block output requires an extra bit-reversal operation. Thus, in many situations, you can increase the speed of the FFT block by selecting the **Output in bit-reversed order** check box.

For more information ordering of the output, see "Linear and Bit-Reversed Output Order".

Scale result by FFT length

When you select this parameter, the block divides the output of the FFT by the FFT length. This option is useful when you want the output of the FFT to stay in the same amplitude range as its input. This is particularly useful when working with fixed-point data types.

Inherit FFT length from input dimensions

Select to inherit the FFT length from the input dimensions. When you select this check box, the input length must be a power of two. When you do not select this check box, the **FFT length** parameter becomes available to specify the length.

FFT length

Specify FFT length. This parameter becomes available only when you do not select the **Inherit FFT length from input dimensions** parameter.

When you set the **FFT implementation** parameter to Radix-2, or when you check the **Output in bit-reversed order** check box, this value must be a power of two.

The **Data Types** pane of the FFT block dialog appears as follows.

Function Block Parameters: FFT

FFT

Compute the fast Fourier transform (FFT) across the first dimension of the input.

When you set the 'FFT implementation' parameter to 'Radix-2', the FFT length must be a power of two.

| Main | Data Types |

Fixed-point operational parameters

Rounding mode: Floor   Overflow mode: Wrap

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block in is floating point, all block data types match the input. When the block input is fixed point, all internal data types a signed fixed point.

	Data Type	Assistant	Minimum	Maximum
Sine table:	Inherit: Same word length as input	>>		
Product output:	Inherit: Inherit via internal rule	>>		
Accumulator:	Inherit: Inherit via internal rule	>>		
Output:	Inherit: Inherit via internal rule	>>	[]	[]

☐ Lock data type settings against changes by the fixed-point tools

OK   Cancel   Help

Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; instead, they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, Inherit: Same word length as input

- An expression that evaluates to a valid data type, for example, fixdt(1,16)

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; instead, they are always saturated and rounded to Nearest.

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Sine table data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Product output data type

Specify the product output data type. See Fixed-Point Data Types on page 586 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See Fixed-Point Data Types on page 586 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See Fixed-Point Data Types on page 586 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.

When you select `Inherit:  Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The equations that the block uses to calculate the ideal output word length and fraction length depend on the setting of the **Divide butterfly outputs by two** check box.
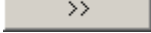
— When you select the **Divide butterfly outputs by two** check box, the ideal output word and fraction lengths are the same as the input word and fraction lengths.

— When you clear the **Divide butterfly outputs by two** check box, the block computes the ideal output word and fraction lengths according to the following equations:

$$WL_{ideal\ output} = WL_{input} + floor(\log_2(FFT\ length - 1)) + 1$$

$$FL_{ideal\ output} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see "Inherit via Internal Rule".

• An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

FFT

Examples

See the section on "Transform Time-Domain Data into Frequency Domain" in the *DSP System Toolbox User's Guide*.

References

[1] Orfanidis, S. J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1996, p. 497.

[2] Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996.

[3] FFTW (http://www.fftw.org)

[4] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

See Also

DCT	DSP System Toolbox
IFFT	DSP System Toolbox
Pad	DSP System Toolbox

fft	MATLAB
ifft	MATLAB
bitrevorder	Signal Processing Toolbox
Simulink Coder	Simulink Coder

Filter Realization Wizard

Purpose   Construct filter realizations using Digital Filter block or Sum, Gain, and Delay blocks

Library   Filtering / Filter Implementations

dsparch4

Description
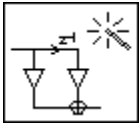
Note Use this block to implement fixed-point or floating-point digital filters using Sum, Gain, and Delay blocks or the Digital Filter block. You can either design a filter by using the block parameters, or import the coefficients of a filter you have designed elsewhere.

The following blocks also implement digital filters, but serve slightly different purposes:

- Digital Filter — Use to implement floating-point or fixed-point filters that you have already designed

- Digital Filter Design — Use to design, analyze, and then implement floating-point filters.

The Filter Realization Wizard is a tool for automatically implementing a digital filter. You must specify a filter, its structure, and the data types for the inputs, outputs, and computations. The filter can support double-precision, single-precision, or fixed-point data types.

The Filter Realization Wizard can implement a digital filter in one of two ways. It can use a Digital Filter block, or it can create a subsystem block that implements the specified filter using Sum, Gain, and Delay blocks. If the Filter Realization Wizard creates a Digital Filter block, double-click the block to open the dialog box. If it creates a subsystem, double-click the subsystem block to see the filter implementation as shown in the figure below.
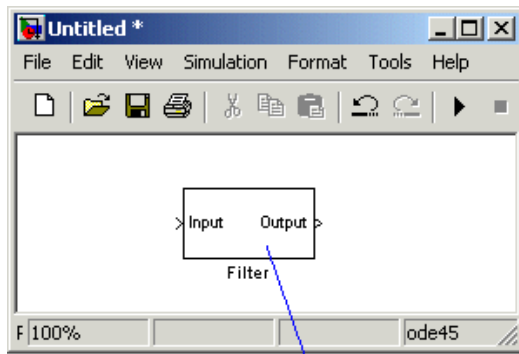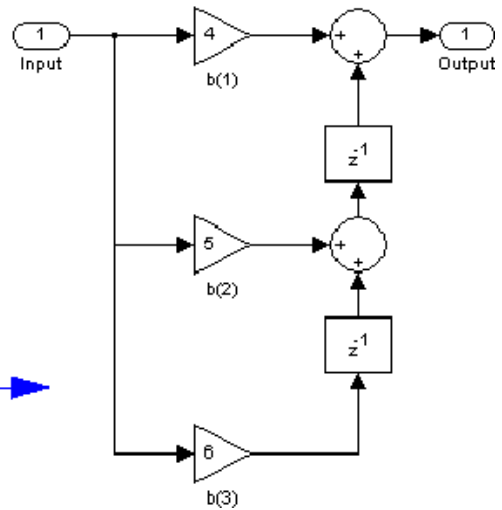
For more information about filter implementation, see "Specify the Filter Implementation" on page 2-601.

The parameters of the Filter Realization Wizard are a part of a larger GUI, the Filter Design and Analysis Tool (fdatool). You can use FDATool to design and analyze your filter, and then use the Filter Realization Wizard parameters to implement the filter in your models.

Specify the Filter and Data Types

To specify a purely double-precision filter, you can either design a filter using the **Design Filter** panel, or import a filter using the **Import Filter** panel. (You can import dfilt filter objects as well as vectors of filter coefficients designed using Signal Processing Toolbox and DSP System Toolbox functions.)

You can also specify a fixed-point filter or a single-precision filter by using the **Set Quantization Parameters** panel.

Filter Realization Wizard

> **Note** *Running* a model containing implementations of fixed-point filters requires the Simulink Fixed Point product, but you can still edit models containing such filter implementations without it. See the Simulink Fixed Point documentation for more information.

See the following topics to learn how to use the panels to specify your filter:

- For more information on the **Design Filter** panel, see "FDATool" in the Signal Processing Toolbox documentation.

- For more information on the **Import Filter** panel, see in the Signal Processing Toolbox documentation.

- For more information on the **Set Quantization Parameters** panel, see "Access the Quantization Features of FDATool".

To open a panel, click the appropriate button in the lower-left corner of FDATool.

Supported Filter Structures

The Filter Realization Wizard supports the following structures:

- Direct form I

- Direct form I, second-order sections

- Direct form I transposed

- Direct form I transposed, second-order sections

- Direct form II

- Direct form II, second-order sections

- Direct form II transposed

- Direct form II transposed, second-order sections

- Direct form FIR

- Direct form FIR transposed

- Direct form symmetric FIR

- Direct form antisymmetric FIR

- Lattice all-pass

- Lattice AR

- Lattice ARMA

- Lattice MA for maximum phase

- Lattice MA for minimum phase

- Cascade

- Parallel

Specify the Filter Implementation

You can determine how the Filter Realization Wizard models the specified filter using the **Build model using basic elements** check box. When you select this check box, the Filter Realization Wizard creates a subsystem block that implements your filter using Sum, Gain, and Delay blocks. When you clear this check box, the Filter Realization Wizard uses a Digital Filter block to implement your filter. The **Build model using basic elements** check box is available only when your filter can be implemented using a Digital Filter block.

The Filter Realization Wizard can generate a subsystem that represents either a double-precision or fixed-point filter. You must install the Simulink Fixed Point product to simulate a fixed-point filter. You can still edit the blocks used to implement the filter without installing the Simulink Fixed Point product.

Filter Realization Wizard

Double-precision filter implemented with Sum, Gain, and Delay blocks
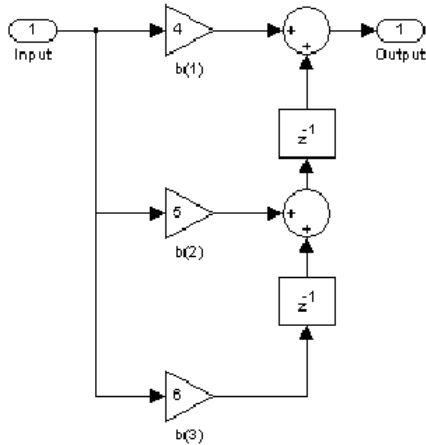
Fixed-point filter implemented with Sum, Gain, Delay, and Conversion blocks
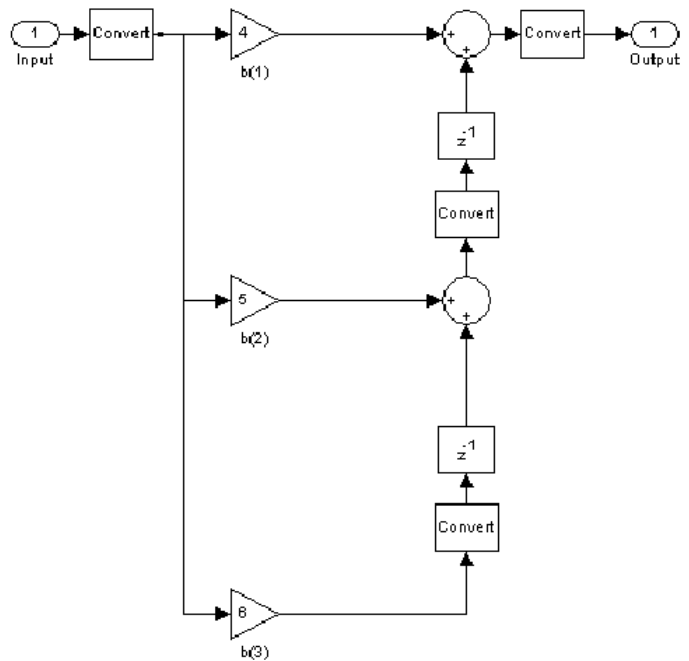
Implementations of Double-Precision and Fixed-Point Filters

Command Line Alternative

The `dfilt` (digital filter) object has a method, `realizemdl`, that allows you to access the capabilities of the Filter Realization Wizard from the command line.

For more information about the `realizemdl` method, see the following sections:

- The topic on "Methods" in the `dfilt` reference page.

- The `realizemdl` reference page.

**Dialog
Box**

Note The following parameters for the Filter Realization Wizard are in the **Realize Model** pane of the Filter Design and Analysis Tool (FDATool) GUI. To open different panels of FDATool, click the different buttons at the lower-left corner. For more information about relevant panels, see "Specify the Filter and Data Types" on page 2-599.

Filter Realization Wizard

Block Name

Enter the name of the new filter block.

Destination

Specify where the new filter block should be created. This can be in a new model or in the current (most recently selected) model.

User Defined

Specify the name of the target subsystem in which the Filter Realization Wizard should create the new filter block.

Overwrite generated block "Filter" block

When selected, the block overwrites any filter block in the current model with the name specified in the **Block Name** parameter. This parameter is enabled when the **Destination** parameter is set to Current.

Build model using basic elements

Select this check box to implement your filter using Sum, Gain, and Delay blocks. Clear this check box to implement your filter using the Digital Filter block. This parameter is available only when your filter can be modeled using the Digital Filter block.

Optimize for zero gains

Select this check box to remove zero-gain paths from the filter structure. For an example, see "Optimize the Filter Structure".

Optimize for unity gains

Select this check box to substitute gains equal to 1 with a wire (short circuit). For an example, see "Optimize the Filter Structure".

Optimize for negative gains

Select this check box to substitute gains equal to -1 with a wire (short circuit), and change the corresponding sums to subtractions. For an example, see "Optimize the Filter Structure".

Optimize delay chains

Select this check box to substitute any delay chains made up of n unit delays with a single delay by n. For an example, see "Optimize the Filter Structure".

Filter Realization Wizard

Optimize for unity scale values

Select this check box to remove all scale value multiplications by 1 from the filter structure.

Input processing

Specify how the generated filter block or subsystem block processes the input. Depending on the type of filter you are designing, one or both of the following options may be available:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

For more information about sample- and frame-based processing, see "Sample- and Frame-Based Concepts".

Rate options

For multirate filters, specify how the block should process the input. You can set this parameter to one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate.

- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples.

Realize Model

Click to create a filter block according to the settings you've specified. When the **Build model using basic elements** check box is selected, the filter is implemented as a subsystem block consisting of Sum, Gain, and Delay blocks. To see the filter implementation, double-click the subsystem block in your model.

> **Note** For more information about relevant parameters in other panels of FDATool, see "Specify the Filter and Data Types" on page 2-599.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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

Supported Data Types
- Double-precision floating point
- Single-precision floating point — Supported only when you install Simulink Fixed Point.
- Fixed point (signed and unsigned) — Supported only when you install Simulink Fixed Point and Fixed-Point Toolbox™.

See Also

Digital Filter	DSP System Toolbox
Digital Filter Design	DSP System Toolbox
filter	DSP System Toolbox
realizemdl	DSP System Toolbox
dfilt	DSP System Toolbox

- "Filter Design"
- "Filter Analysis"
- "Select a Filter Design Block"

FIR Decimation

Purpose	Filter and downsample input signals
Library	Filtering / Multirate Filters
	dspmlti4

Description
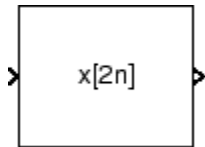
The FIR Decimation block resamples the discrete-time input at a rate K times slower than the input sample rate, where K is the integer value you specify for the **Decimation factor** parameter. To do so, the block implements a polyphase filter structure and performs the following operations:

1 Filters the data in each channel of the input using a direct-form FIR filter.

2 Downsamples each channel of filtered data by discarding $K-1$ consecutive samples following each sample that is retained.

The block uses a polyphase filter implementation because it is more efficient than straightforward filter-then-decimate algorithms. See Fliege [1] for more information.

You can use the FIR Decimation block inside triggered subsystems when you set the **Rate options** parameter to Enforce single-rate processing.

Specifying the Filter Coefficients

The FIR Decimation block can operate in two different modes. Select the mode in the **Coefficient source** group box. If you select:

- **Dialog parameters**, you enter information about the filter such as structure and coefficients in the block dialog box.

- **Multirate filter object (MFILT)**, you specify the filter using an mfilt object.

When you select **Dialog parameters**, you use the **FIR filter coefficients** parameter to specify the numerator coefficients of the FIR filter transfer function $H(z)$.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

You can generate the FIR filter coefficient vector, `[b(1) b(2) ... b(m)]`, using one of the Signal Processing Toolbox filter design functions (such as `intfilt`).

The filter you specify should be a lowpass filter with a normalized cutoff frequency no greater than $1/K$. The block internally initializes all filter states to zero.

Note You can use the block method to create your own filter blocks directly from your `mfilt` objects.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block resamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block resamples the input:

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. To decimate the output while maintaining the input sample rate, the block resamples the data in each column of the input such that the frame size of the output (M_o) is $1/K$ times that of the input ($M_o = M_i/K$),

  In this mode, the input frame size, M_i, must be a multiple of the **Decimation factor**, K.

  For an example of single-rate FIR Decimation, see "Example 1 — Single-Rate Processing" on page 2-612.

FIR Decimation

- When you set the **Rate options** parameter to `Allow multirate processing`, the input and output of the FIR Decimation block are the same size, but the sample rate of the output is *K* times slower than that of the input. In this mode, the block treats an M_i-by-*N* matrix input as *N* independent channels. The block decimates each column of the input over time by keeping the frame size constant ($M_i=M_o$), and making the output frame period (T_{fo}) *K* times longer than the input frame period ($T_{fo} = K*T_{fi}$).

  See "Example 2— Multirate Frame-Based Processing" on page 2-612 for an example that uses the FIR Decimation block in this mode.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an *M*-by-*N* matrix input as *M*N* independent channels, and decimates each channel over time. The output sample period (T_{so}) is *K* times longer than the input sample period ($T_{so} = K*T_{si}$), and the input and output sizes are identical.

Latency

When you use the FIR Decimation block in sample-based processing mode, the block always has zero-tasking latency. *Zero-tasking latency* means that the block propagates the first filtered input sample (received at time *t*= 0) as the first output sample. That first output sample is then followed by filtered input samples *K*+1, 2*K*+1, and so on.

When you use the FIR Decimation block in frame-based processing mode with a frame size greater than one, the block may exhibit *one-frame latency*. Cases of one-frame latency can occur when the input frame size is greater than one, and you set the **Input processing** and **Rate options** parameters of the FIR Decimation block as follows:

- **Input processing** = `Columns as channels (frame based)`

- **Rate options** = `Allow multirate processing`

In cases of one-frame latency, you can define the value of the first M_i output rows by setting the **Output buffer initial conditions**

parameter. The default value of the **Output buffer initial conditions** parameter is 0. However, you can enter a matrix containing one value for each channel of the input, or a scalar value to be applied to all channels. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample M_i+ 1. That sample is then followed by filtered input samples K+ 1, $2K$+ 1, and so on.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" in the *DSP System Toolbox User's Guide* and "Scheduling" in the *Simulink Coder User's Guide*.

Fixed-Point Data Types

The following diagram shows the data types used within the FIR Decimation block for fixed-point signals.
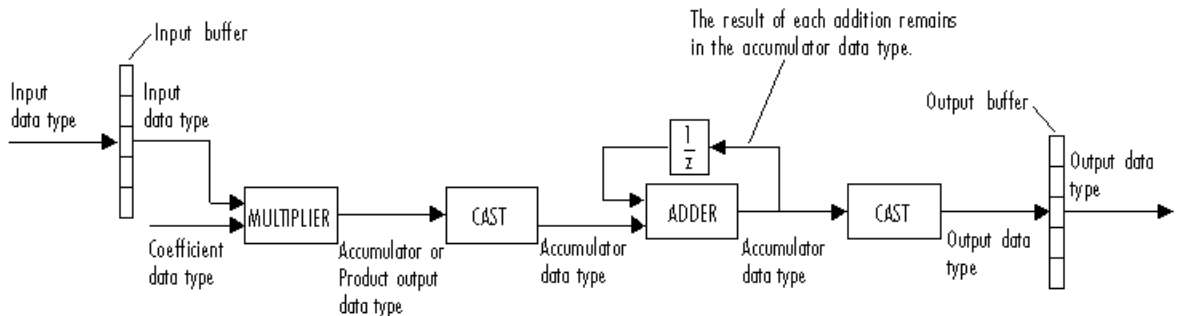
You can set the coefficient, product output, accumulator, and output data types in the block dialog box as discussed in the "Dialog Box" on page 2-613 section. This diagram shows that data is stored in the input buffer with the same data type and scaling as the input. The block stores filtered data and any initial conditions in the output buffer using the output data type and scaling that you set in the block dialog box.

When at least one of the inputs to the multiplier is real, the output of the multiplier is in the product output data type. When both inputs

FIR Decimation

to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed by this block, see "Multiplication Data Types" in the *DSP System Toolbox User's Guide*.

Note When the block input is fixed point, all internal data types are signed fixed-point values.

Examples

Example 1 — Single-Rate Processing

In the ex_firdecimation_ref2 model, the FIR Decimation block decimates a single-channel input with a frame size of 64. Because the block is doing single-rate processing and the **Decimation factor** parameter is set to 4, the output of the FIR Decimation block has a frame size of 16. As shown in the following figure, the input and output of the FIR Decimation block have the same sample rate.
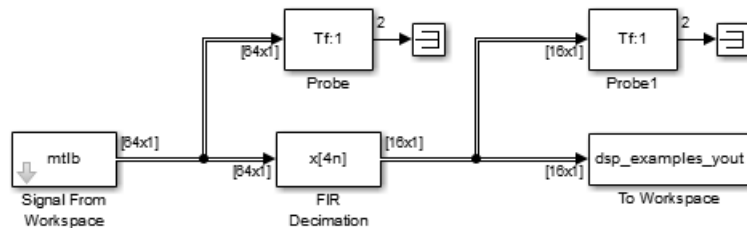
Example 2— Multirate Frame-Based Processing

In the ex_firdecimation_ref1 model, the FIR Decimation block decimates a single-channel input with a frame period of one second. Because the block is doing multirate frame-based processing and the **Decimation factor** parameter is set to 4, the frame period of the output is four seconds. As shown in the following figure, the input and output of the FIR Decimation block have the same frame size, but the sample rate of the output is four times that of the input.
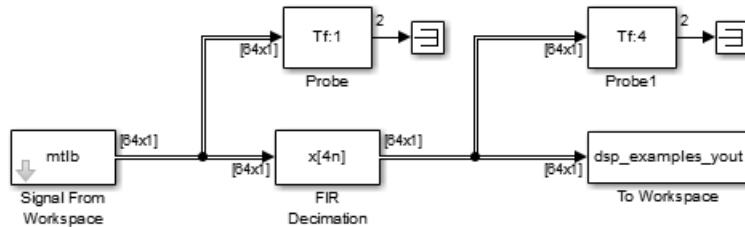
Example 3

The ex_polyphasedec model illustrates the underlying polyphase implementations of the FIR Decimation block. Run the model, and view the results on the scope. The output of the FIR Decimation block matches the output of the Polyphase Decimation Filter block.

Example 4

The ex_mrf_nlp model illustrates the use of the FIR Decimation block in a number of multistage multirate filters.

Dialog Box

Coefficient Source

The FIR Decimation block can operate in two different modes. Select the mode in the **Coefficient source** group box. If you select

- **Dialog parameters**, you enter information about the filter such as structure and coefficients in the block mask.

- **Multirate filter object (MFILT)**, you specify the filter using an mfilt object.

Different items appear on the FIR Decimation block dialog depending on whether you select **Dialog parameters** or **Multirate filter object (MFILT)** in the **Coefficient source** group box. See the following sections for details:

- "Specify Filter Characteristics in Dialog" on page 2-614

- "Specify Multirate Filter Object" on page 2-622

FIR Decimation

Specify Filter Characteristics in Dialog

The **Main** pane of the FIR Decimation block dialog appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.

Function Block Parameters: FIR Decimation

FIR Decimation

Apply an FIR filter to the input signal, then downsample the signal by an integer-valued factor. The block imp
FIR filter using a polyphase filter structure.

Coefficient source

◉ Dialog parameters

○ Multirate filter object (MFILT)

| Main | Data Types |

Parameters

FIR filter coefficients: fir1(35,0.4)

Decimation factor: 2

Filter structure: Direct form

Input processing: Columns as channels (frame based)

Rate options: Enforce single-rate processing

View Filter

OK   Cancel   Help

FIR Decimation

FIR filter coefficients

Specify the lowpass FIR filter coefficients, in descending powers of *z*.

Decimation factor

Specify the integer factor, *K*, by which to decrease the sample rate of the input sequence.

Filter Structure

Choose whether to implement a `Direct form` or `Direct form transposed` filter.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

Specify the method by which the block should decimate the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate and decimates the signal by decreasing the output frame size by a factor of *K*. To select this option, you must set the **Input**

processing parameter to Columns as channels (frame based).

- Allow multirate processing — When you select this option, the block decimates the signal such that the output sample rate is *K* times slower than the input sample rate.

Output buffer initial conditions

In the case of *one-frame latency*, this parameter specifies the output of the block until the first filtered input sample is available. The default value of this parameter is 0, but you can enter a matrix containing one value for each channel, or a scalar value to be applied to all signal channels. This parameter appears only when you configure the block to perform multirate processing.

See "Latency" on page 2-610 for more information about latency in the FIR Decimation block.

View filter response

This button opens the Filter Visualization Tool (fvtool) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note If you specify a filter in the **Multirate filter variable** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

The **Data Types** pane of the FIR Decimation block dialog appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.

FIR Decimation

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to Nearest.

Note The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is Inherit: Inherit via internal rule

- **Accumulator data type** is Inherit: Inherit via internal rule

- **Output data type** is Inherit: Same as accumulator

With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See "Fixed-Point Data Types" on page 2-611 and "Multiplication Data Types" for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same word length as input

- An expression that evaluates to a valid data type, for example, fixdt(1,16,0)

FIR Decimation

Click the **Show data type assistant** button $\boxed{\quad \gg \quad}$ to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-611 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button $\boxed{\quad \gg \quad}$ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-611 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to
display the **Data Type Assistant**, which helps you set the
Accumulator data type parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink
User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on
page 2-611 for illustrations depicting the use of the output data
type in this block. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same
  as accumulator

- An expression that evaluates to a valid data type, for example,
  fixdt(1,16,0)

Click the **Show data type assistant** button ⟩⟩ to
display the **Data Type Assistant**, which helps you set the
Output data type parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide*
for more information.

Minimum

Specify the minimum value that the block should output. The
default value is [] (unspecified). Simulink software uses this
value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The
default value is [] (unspecified). Simulink software uses this
value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
> Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Specify Multirate Filter Object

The **Main** pane of the FIR Decimation block dialog appears as follows when you select **Multirate filter object (MFILT)** in the **Coefficient source** group box.

FIR Decimation

Multirate filter variable

Specify the multirate filter object (`mfilt`) that you would like the block to implement. You can do this in one of three ways:

- You can fully specify the `mfilt` object in the block mask.

- You can enter the variable name of a `mfilt` object that is defined in any workspace.

- You can enter a variable name for a `mfilt` object that is not yet defined, as shown in the default value.

For more information on creating `mfilt` objects, see the `mfilt` function reference page.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

Specify the method by which the block should decimate the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and

decimates the signal by decreasing the input frame size by a factor of *K*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.

- `Allow multirate processing` — When you select this option, the block decimates the sample rate such that the output sample rate is *K* times slower than the input sample rate.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the `mfilt` object specified in the **Multirate filter variable** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note If you specify a filter in the **Multirate filter variable** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

The **Data Types** pane of the FIR Decimation block dialog appears as follows when you select **Multirate filter object (MFILT)** in the **Coefficient source** group box.

FIR Decimation

Apply an FIR filter to the input signal, then downsample the signal by an integer-valued factor. The block impleme
FIR filter using a polyphase filter structure.

Coefficient source

○ Dialog parameters

◉ Multirate filter object (MFILT)

| Main | Data Types |

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block inp
floating point, all block data types match the input. When the block input is fixed point, all internal data types are
fixed point.

Fixed-point operational parameters

Rounding mode: Floor   Overflow mode: Wrap

Fixed-point data types

	Data Type
Coefficients	Same word length as input
Product output	Same as input
Accumulator	Same as product output
Output	Same as accumulator

[OK]   [Cancel]   [Help]

The fixed-point settings of the filter object specified on the **Main** pane are displayed on the **Data Types** pane. You cannot change these settings directly on the block mask. To change the fixed-point settings you must edit the filter object directly.

For more information on multirate filter objects, see the `mfilt` function reference page.

References [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Downsample	DSP System Toolbox
FIR Interpolation	DSP System Toolbox
FIR Rate Conversion	DSP System Toolbox
decimate	Signal Processing Toolbox

FIR Decimation

fir1	Signal Processing Toolbox
fir2	Signal Processing Toolbox
firls	Signal Processing Toolbox

Purpose   Model FIR filters

Library   Filtering / Filter Implementations

                  `dsparch4`

Description   This block is the same as the Simulink Discrete FIR Filter block. For more information, see the Discrete FIR Filter reference page in the Simulink documentation.

FIR Interpolation

Purpose   Upsample and filter input signals

Library   Filtering / Multirate Filters

dspmlti4
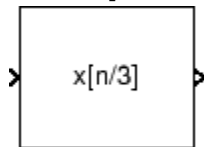
Description   The FIR Interpolation block resamples the discrete-time input at a
rate L times faster than the input sample rate, where L is the integer
value you specify for the **Interpolation factor** parameter. To do so,
the block implements a polyphase filter structure and performs the
following operations:

x[n/3]

- Upsamples each channel of the input to a higher rate by inserting
  L–1 zeros between samples.

- Filters each channel of the upsampled data using a direct-form FIR
  filter.

The block uses a polyphase filter implementation because it is more
efficient than straightforward upsample-then-filter algorithms. See
Fliege [1] for more information.

You can use the FIR Interpolation block inside triggered subsystems
when you set the **Rate options** parameter to Enforce single-rate
processing.

Specifying the Filter Coefficients

The FIR Interpolation block can operate in two different modes. Select
the mode in the **Coefficient source** group box. If you select:

- **Dialog parameters**, you enter information about the filter such as
  structure and coefficients in the block dialog box.

- **Multirate filter object (MFILT)**, you specify the filter using an
  mfilt object.

When you select **Dialog parameters**, you use the **FIR filter
coefficients** parameter to specify the numerator coefficients of the
FIR filter transfer function $H(z)$.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

You can generate the FIR filter coefficient vector, [b(1) b(2) ... b(m)], using one of the Signal Processing Toolbox filter design functions (such as intfilt).

The filter you specify should be a lowpass filter with a length greater than the interpolation factor (m>L) and a normalized cutoff frequency no greater than 1/L. The block internally initializes all filter states to zero.

Note You can use the block method to create your own filter blocks directly from your mfilt objects.

Frame-Based Processing

When you set the **Input processing** parameter to Columns as channels (frame based), the block resamples each column of the input over time. In this mode, the block can perform either single-rate or multirate processing. You can use the **Rate options** parameter to specify how the block resamples the input:

- When you set the **Rate options** parameter to Enforce single-rate processing, the input and output of the block have the same sample rate. To interpolate the output while maintaining the input sample rate, the block resamples the data in each column of the input such that the frame size of the output (M_o) is L times larger than that of the input ($M_o = M_i * L$).

  For an example of single-rate FIR Interpolation, see "Example 1 — Single-Rate Processing" on page 2-634.

- When you set the **Rate options** parameter to Allow multirate processing, the input and output of the FIR Interpolation block are the same size. However, the sample rate of the output is L times faster than that of the input. In this mode, the block treats an M_i-by-N matrix input as N independent channels. The block

interpolates each column of the input over time by keeping the frame size constant ($M_i = M_o$), while making the output frame period (T_{fo}) L times shorter than the input frame period ($T_{fo} = T_{fi}/L$).

See "Example 2 — Multirate Frame-Based Processing" on page 2-635 for an example that uses the FIR Interpolation block in this mode.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-N matrix input as $M*N$ independent channels, and interpolates each channel over time. The output sample period (T_{so}) is L times shorter than the input sample period ($T_{so} = T_{si}/L$), while the input and output sizes remain identical.

Latency

When you run your models in Simulink `SingleTasking` mode or set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`, the FIR Interpolation block always has zero-tasking latency. *Zero-tasking latency* means that the block propagates the first filtered input sample (received at time t=0) as the first output sample. That first output sample is then followed by L–1 interpolated values, the second filtered input sample, and so on.

The only time the FIR Interpolation block exhibits latency is when you set the **Rate options** parameter set to `Allow multirate processing` and run your models in Simulink `MultiTasking` mode. The amount of latency for multirate, multitasking operation depends on the setting of the **Input processing** parameter, as shown in the following table.

Input processing	Latency
`Elements as channels (sample based)`	L samples
`Columns as channels (frame based)`	L frames (M_i samples per frame)

When the block exhibits latency, the default initial condition is zero. Alternatively, you can use the **Output buffer initial conditions** parameter to specify a matrix of initial conditions containing one value for each channel or a scalar initial condition to be applied to all channels. The block scales the **Output buffer initial conditions** by the **Interpolation factor** and outputs the scaled initial conditions until the first filtered input sample becomes available.

When the block is in sample-based processing mode, the block outputs the scaled initial conditions at the start of each channel, followed immediately by the first filtered input sample, then $L–1$ interpolated values, and so on.

When the block is in frame-based processing mode and using the default initial condition of zero, the first M_i*L 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*L+1. That value is then followed by $L–1$ interpolated values, the second filtered input sample, and so on.


Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" in the *DSP System Toolbox User's Guide* and "Scheduling" in the *Simulink Coder User's Guide*.

Fixed-Point Data Types

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

FIR Interpolation
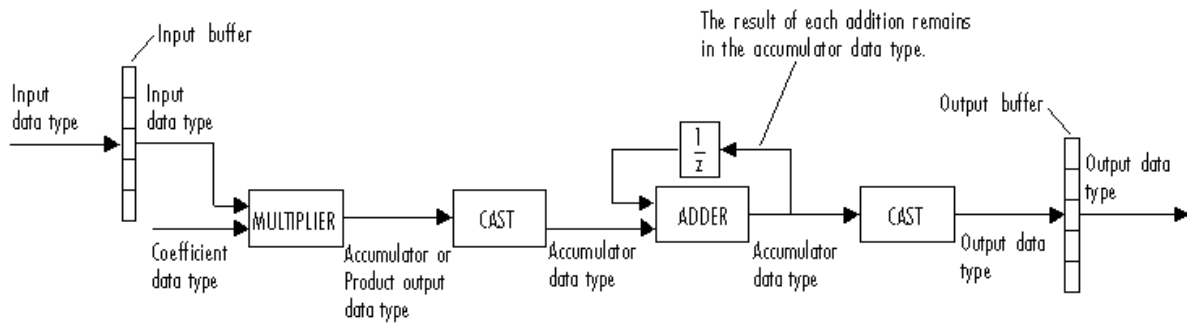
You can set the coefficient, product output, accumulator, and output data types in the block dialog as discussed in "Dialog Box" on page 2-636 section. This diagram shows that input data is stored in the input buffer with the same data type and scaling as the input. The block stores filtered data and any initial conditions in the output buffer using the output data type and scaling that you set in the block dialog box.

When at least one of the inputs to the multiplier is real, the output of the multiplier is in the product output data type. When both inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed by this block, see "Multiplication Data Types" in the *DSP System Toolbox User's Guide*.

**Note** When the block input is fixed point, all internal data types are signed fixed point.

Examples

Example 1 — Single-Rate Processing

In the ex_firinterpolation_ref2, the FIR Interpolation block interpolates a single-channel input with a frame size of 16. Because the block is doing single-rate processing and the **Interpolation factor** parameter is set to 4, the output of the FIR Interpolation block has a frame size of 64. As shown in the following figure, the input and output of the FIR Interpolation block have the same sample rate.
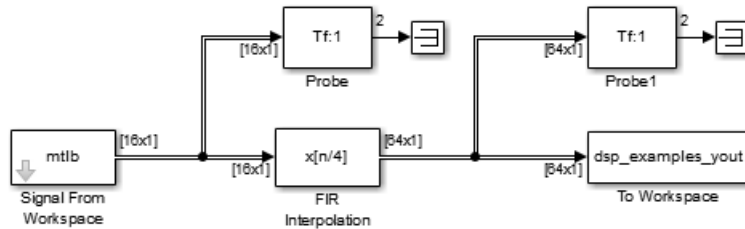
Example 2 — Multirate Frame-Based Processing

In the ex_firinterpolation_ref1, the FIR Interpolation block interpolates a single-channel input with a frame period of 1 second (**Sample time** = 1/64 and **Samples per frame** = 64). Because the block is doing multirate frame-based processing and the **Interpolation factor** parameter is set to 4, the output of the FIR Interpolation block has a frame period of 0.25 seconds. As shown in the following figure, the input and output of the FIR Interpolation block have the same frame size, but the sample rate of the output is 1/4 times that of the input.
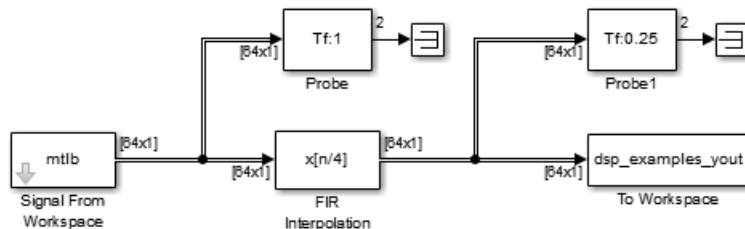
Example 3

The ex_polyphaseinterp model illustrates the underlying polyphase implementations of the FIR Interpolation block. Run the model, and view the results on the scope. The output of the FIR Interpolation block matches the output of the Polyphase Interpolation Filter block.

Example 4

The ex_mrf_nlp model illustrates the use of the FIR Interpolation block in a number of multistage multirate filters.

FIR Interpolation

Dialog Box

Coefficient Source

The FIR Interpolation block can operate in two different modes. Select the mode in the **Coefficient source** group box. If you select

- **Dialog parameters**, you enter information about the filter such as structure and coefficients in the block mask.

- **Multirate filter object (MFILT)**, you specify the filter using an `mfilt` object.

Different items appear on the FIR Interpolation block dialog depending on whether you select **Dialog parameters** or **Multirate filter object (MFILT)** in the **Coefficient source** group box. See the following sections for details:
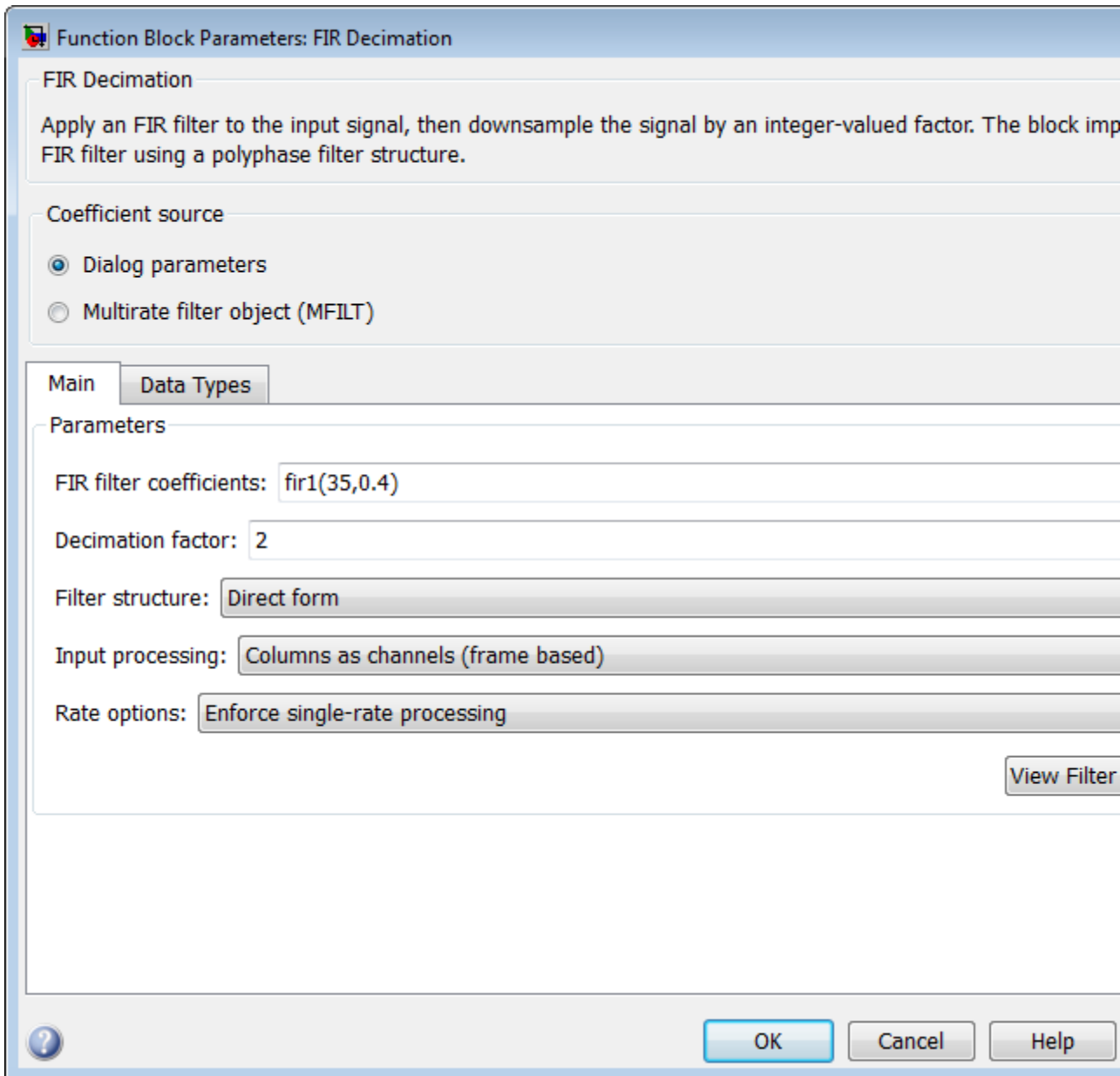
- "Specify Filter Characteristics in Dialog" on page 2-636

- "Specify Multirate Filter Object" on page 2-645

Specify Filter Characteristics in Dialog

The **Main** pane of the FIR Interpolation block dialog appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.

Function Block Parameters: FIR Interpolation

FIR Interpolation

Upsample the input signal by an integer-valued factor, then apply an FIR filter. The block scales the filter coef the interpolation factor and implements the FIR filter using a polyphase structure.

Coefficient source

◉ Dialog parameters

○ Multirate filter object (MFILT)

| Main | Data Types |

Parameters

FIR filter coefficients: `fir1(15,1/4)`

Interpolation factor: `3`

Input processing: `Columns as channels (frame based)`

Rate options: `Enforce single-rate processing`

View Filter

OK   Cancel   Help

FIR Interpolation

FIR filter coefficients

Specify the FIR filter coefficients, in descending powers of *z*.

Interpolation factor

Specify the integer factor, *L*, by which to increase the sample rate of the input sequence.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

> **Note** The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

Specify the method by which the block should interpolate the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and interpolates the signal by increasing the output frame size by a factor of *L*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.

- `Allow multirate processing` — When you select this option, the block interpolates the signal such that the output sample rate is *L* times faster than the input sample rate.

Output buffer initial conditions

> In cases of nonzero latency, the block divides this parameter by the **Interpolation factor** and outputs the results at the output port until the first filtered input sample is available. The default initial condition value is 0, but you can enter a matrix containing one value for each channel, or a scalar to be applied to all signal channels. This parameter appears only when you configure the block to perform multirate processing.
>
> Output buffer initial conditions are stored in the output data type and scaling.
>
> See "Latency" on page 2-632 for more information about latency in the FIR Interpolation block.

View filter response

> This button opens the Filter Visualization Tool (fvtool) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note If you specify a filter in the **Multirate filter variable** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

FIR Interpolation

The **Data Types** pane of the FIR Interpolation block dialog appears as follows when you select **Dialog parameters** in the **Coefficient source** group box.

Function Block Parameters: FIR Interpolation

FIR Interpolation

Upsample the input signal by an integer-valued factor, then apply an FIR filter. The block scales the filter coef
the interpolation factor and implements the FIR filter using a polyphase structure.

Coefficient source

◉ Dialog parameters

◯ Multirate filter object (MFILT)

| Main | Data Types |

Fixed-point operational parameters

Rounding mode: Floor ▼   Overflow mode: Wrap

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the blo
is floating point, all block data types match the input. When the block input is fixed point, all internal data typ
signed fixed point.

	Data Type	Assistant	Minimum	Maximum
Coefficients:	Inherit: Same word length as input ▼	>>	[]	[]
Product output:	Inherit: Inherit via internal rule ▼	>>		
Accumulator:	Inherit: Inherit via internal rule ▼	>>		
Output:	Inherit: Same as accumulator ▼	>>	[]	[]

☐ Lock data type settings against changes by the fixed-point tools

[OK]   [Cancel]   [Help]

Rounding mode

    Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Note The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when all the following conditions exist:

- **Product output data type** is `Inherit:  Inherit via internal rule`

- **Accumulator data type** is `Inherit:  Inherit via internal rule`

- **Output data type** is `Inherit:  Same as accumulator`

With these data type settings, the block is effectively operating in full precision mode.
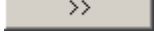

Overflow mode

    Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

    Specify the coefficients data type. See "Fixed-Point Data Types" on page 2-633 and "Multiplication Data Types" for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit:  Same word length as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to
display the **Data Type Assistant**, which helps you set the
Coefficients data type parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-633 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to
display the **Data Type Assistant**, which helps you set the
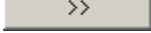**Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-633 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`
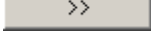
Click the **Show data type assistant** button ⟨ >> ⟩ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-633 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟨ >> ⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:
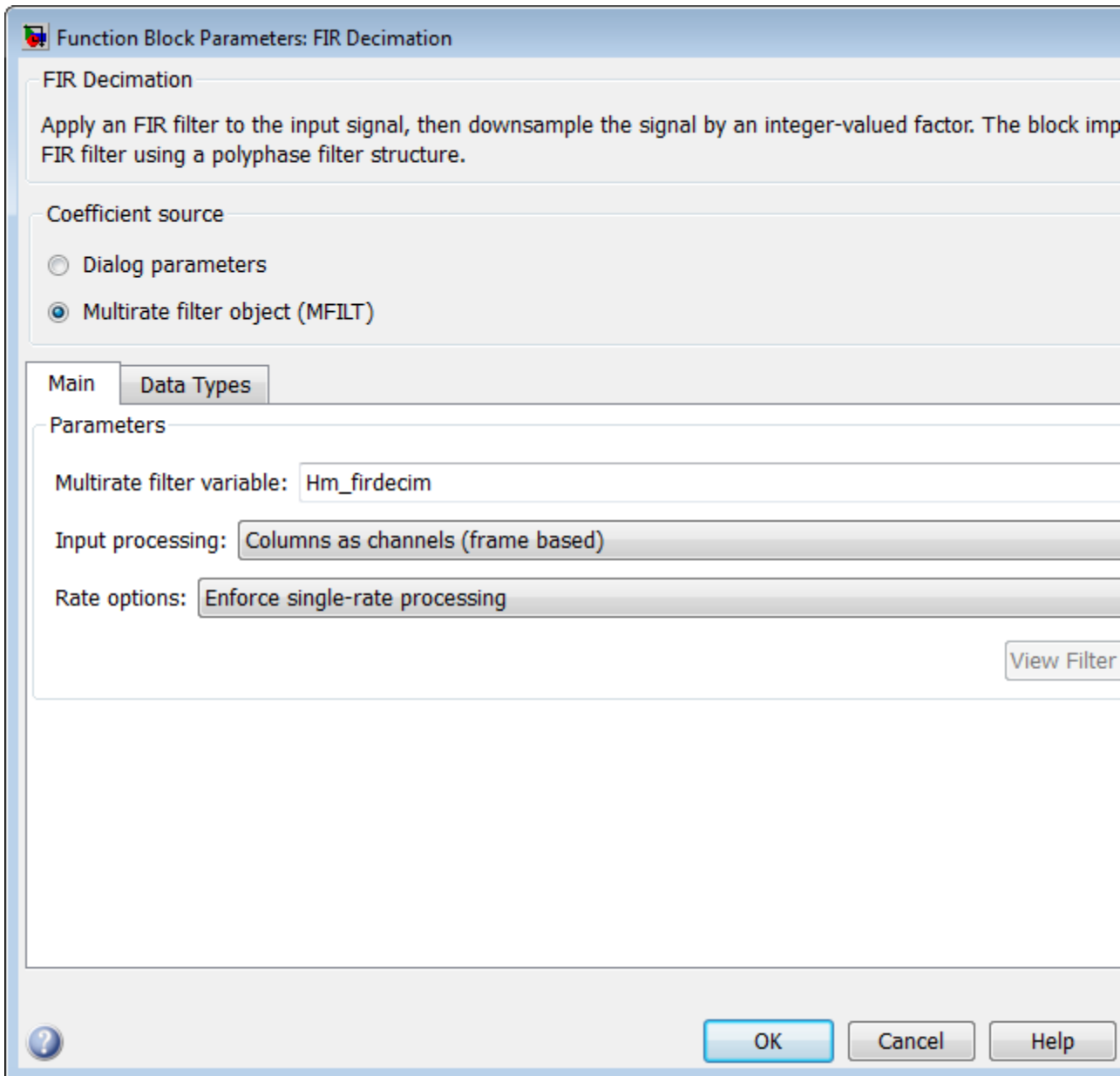
- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
   Select this parameter to prevent the fixed-point tools from
   overriding the data types you specify on the block mask.

Specify Multirate Filter Object

The **Main** pane of the FIR Interpolation block dialog appears as follows
when you select **Multirate filter object (MFILT)** in the **Coefficient
source** group box.

FIR Interpolation

Function Block Parameters: FIR Interpolation

FIR Interpolation

Upsample the input signal by an integer-valued factor, then apply an FIR filter. The block scales the filter coefficie
the interpolation factor and implements the FIR filter using a polyphase structure.

Coefficient source

○ Dialog parameters

● Multirate filter object (MFILT)

| Main | Data Types |

Parameters

Multirate filter variable: Hm_firinterp

Input processing: Columns as channels (frame based)

Rate options: Enforce single-rate processing

View Filter Res

OK   Cancel   Help

Multirate filter variable

Specify the multirate filter object (`mfilt`) that you would like the block to implement. You can do this in one of three ways:

- You can fully specify the `mfilt` object in the block mask.

- You can enter the variable name of a `mfilt` object that is defined in any workspace.

- You can enter a variable name for a `mfilt` object that is not yet defined, as shown in the default value.

For more information on creating `mfilt` objects, see the `mfilt` function reference page.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

Specify the method by which the block should interpolate the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and interpolates the signal by increasing the output frame size by

a factor of *L*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.

- `Allow multirate processing` — When you select this option, the block interpolates the signal such that the output sample rate is *L* times faster than the input sample rate.

View filter response
> This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the `mfilt` object specified in the **Multirate filter variable** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.
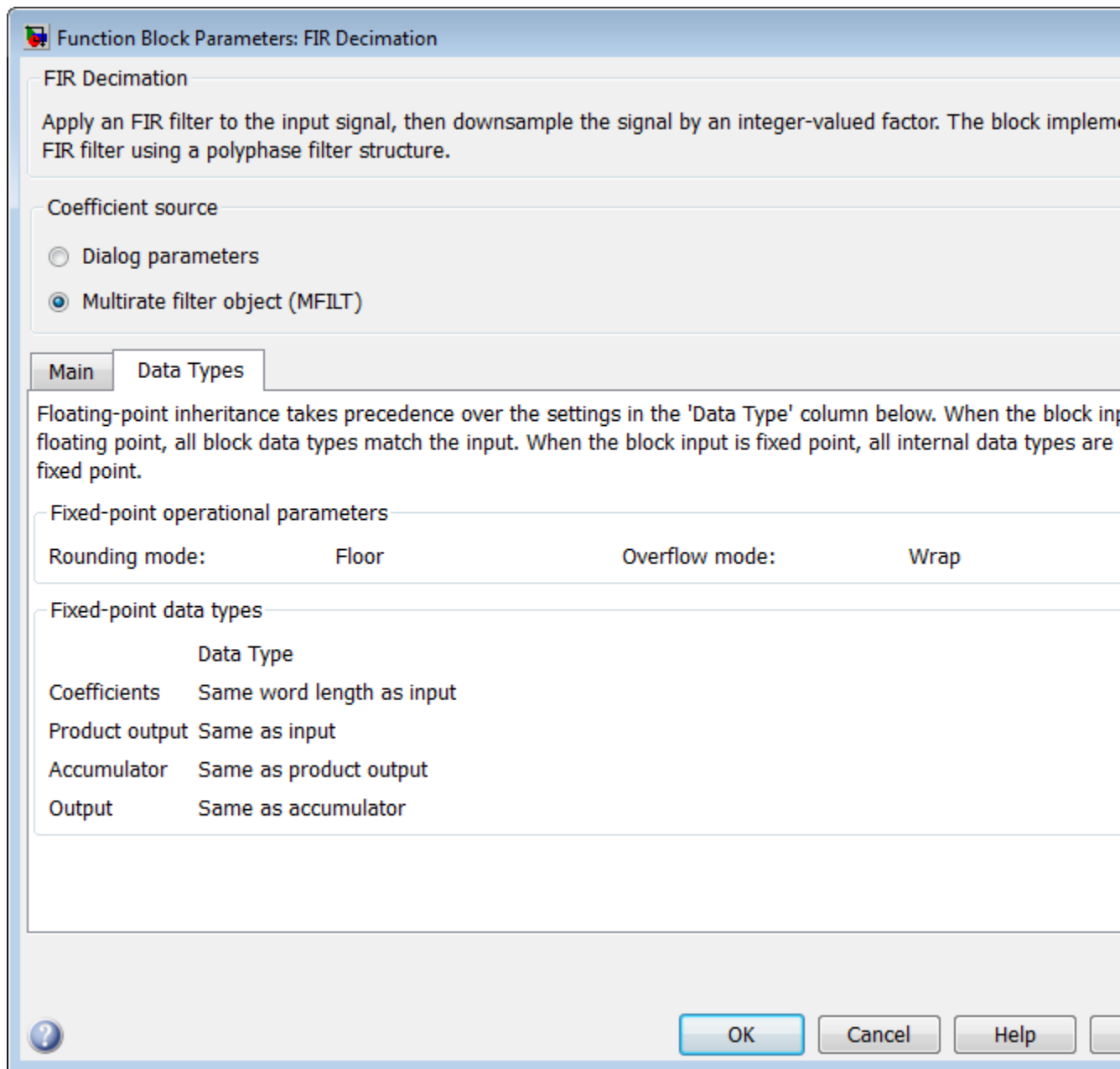

Note If you specify a filter in the **Multirate filter variable** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

The **Data Types** pane of the FIR Interpolation block dialog appears as follows when you select **Multirate filter object (MFILT)** in the **Coefficient source** group box.

Function Block Parameters: FIR Interpolation

FIR Interpolation

Upsample the input signal by an integer-valued factor, then apply an FIR filter. The block scales the filter coef
the interpolation factor and implements the FIR filter using a polyphase structure.

Coefficient source

○ Dialog parameters

● Multirate filter object (MFILT)

| Main | Data Types |

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the bloc
floating point, all block data types match the input. When the block input is fixed point, all internal data types
fixed point.

Fixed-point operational parameters

Rounding mode: Floor   Overflow mode: Wrap

Fixed-point data types

	Data Type
Coefficients	Same word length as input
Product output	Same as input
Accumulator	Same as product output
Output	Same as accumulator

OK   Cancel   Help

FIR Interpolation

The fixed-point settings of the filter object specified on the **Main** pane are displayed on the **Data Types** pane. You cannot change these settings directly on the block mask. To change the fixed-point settings you must edit the filter object directly.

For more information on multirate filter objects, see the `mfilt` function reference page.

References [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets.* West Sussex, England: John Wiley & Sons, 1994.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

FIR Decimation	DSP System Toolbox
FIR Rate Conversion	DSP System Toolbox
Upsample	DSP System Toolbox
fir1	Signal Processing Toolbox

fir2	Signal Processing Toolbox
firls	Signal Processing Toolbox
interp	Signal Processing Toolbox

FIR Rate Conversion

Purpose   Upsample, filter, and downsample input signals

Library   Filtering / Multirate Filters

dspmlti4

Description
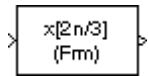
The FIR Rate Conversion block resamples the discrete-time input such that its sample period is K/L times the input sample period (T_{si}). K is the integer value you specify for the **Decimation factor** parameter, and L is the integer value you specify for the **Interpolation factor** parameter.

The block treats each column of the input as a separate channel, and resamples the data in each channel independently over time. To do so, the block implements a polyphase filter structure and performs the following operations:

1 Upsamples the input to a higher rate by inserting L-1 zeros between input samples.

2 Passes the upsampled data through a direct-form II transpose FIR filter.

3 Downsamples the filtered data to a lower rate by discarding K-1 consecutive samples following each sample that the block retains.

The polyphase filter implementation is more efficient than a straightforward upsample-filter-decimate algorithm. See Orfanidis [1] for more information.

Specifying the Resampling Rate

You specify the resampling rate of the FIR Rate Conversion block using the **Decimation factor** and **Interpolation factor** parameters. For an M_i-by-N matrix input, the **Decimation factor**, K, and the **Interpolation factor**, L, must satisfy the following requirements:

- K and L must be relatively prime integers; that is, the ratio K/L cannot be reduced to a ratio of smaller integers.

- $\dfrac{K}{L} = \dfrac{M_i}{M_o}$, where M_{i} and M_{o} are the integer frame sizes of the input and output, respectively.

You can satisfy the second requirement by setting the **Decimation factor**, K, equal to the input frame size, M_{i}. When you do so, the output frame size, M_{o}, equals the **Interpolation factor**, L.

By changing the frame size in this way, the block is able to hold the frame period constant ($T_{\mathrm{fi}} = T_{\mathrm{fo}}$) and achieve the desired conversion of the sample period, such that

$$T_{so} = \frac{K}{L} \times T_{si}$$

where T_{so} is the output sample period.

Specifying the FIR Filter Coefficients

You can specify the FIR filter coefficients in one of two ways:

- To specify the coefficients on the dialog, select the **Dialog parameters** option in the **Coefficient source** group box.

- To specify the coefficients using an `mfilt` object, select the **Multirate filter object (MFILT)** option in the **Coefficient source** group box.

When you select the **Dialog parameters** option, you use the **FIR filter coefficients** parameter to specify the numerator coefficients of the FIR filter transfer function $H(z)$.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

You can generate the FIR filter coefficient vector, `[b(1) b(2) ... b(m)]`, using one of the Signal Processing Toolbox filter design functions (such as `intfilt`).

The coefficient vector you specify must have a length greater than the interpolation factor ($m > L$). The FIR filter must be a lowpass filter with

FIR Rate Conversion

a normalized cutoff frequency, no greater than $\min(1/L, 1/K)$. The block internally initializes all filter states to zero.

Fixed-Point Data Types

The following diagram shows the data types used within the FIR Rate Conversion block for fixed-point signals.
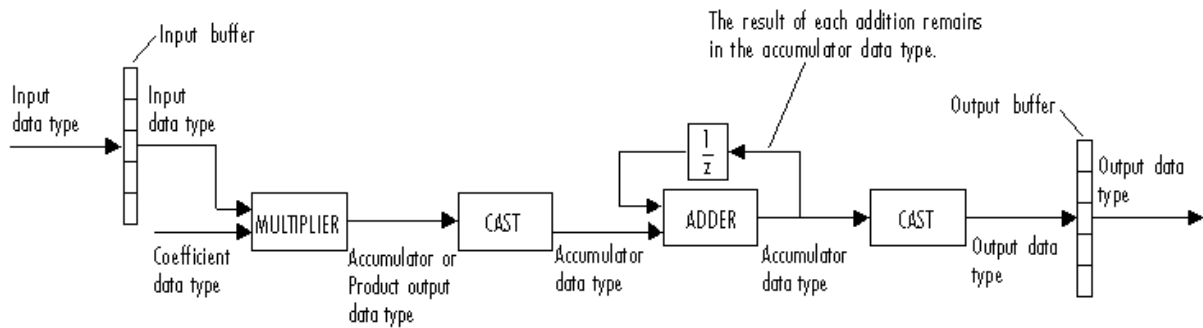
You can set the coefficient, product output, accumulator, and output data types in the block dialog box as discussed in "Dialog Box" on page 2-655. The diagram shows that input data is stored in the input buffer in the same data type and scaling as the input. Filtered data resides in the output buffer in the output data type and scaling that you set in the block dialog. The block stores any initial conditions in the output buffer using the output data type and scaling that you set in the block dialog box.

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types".

Note When the block input is fixed point, all internal data types are signed fixed point.

Examples

Example 1

The following figure shows how the FIR Rate Conversion block converts a 4-by-1 input with a sample period of 3/4, to a 3-by-1 output with a sample period of 1. The frame period (T_f) of 3 remains constant.
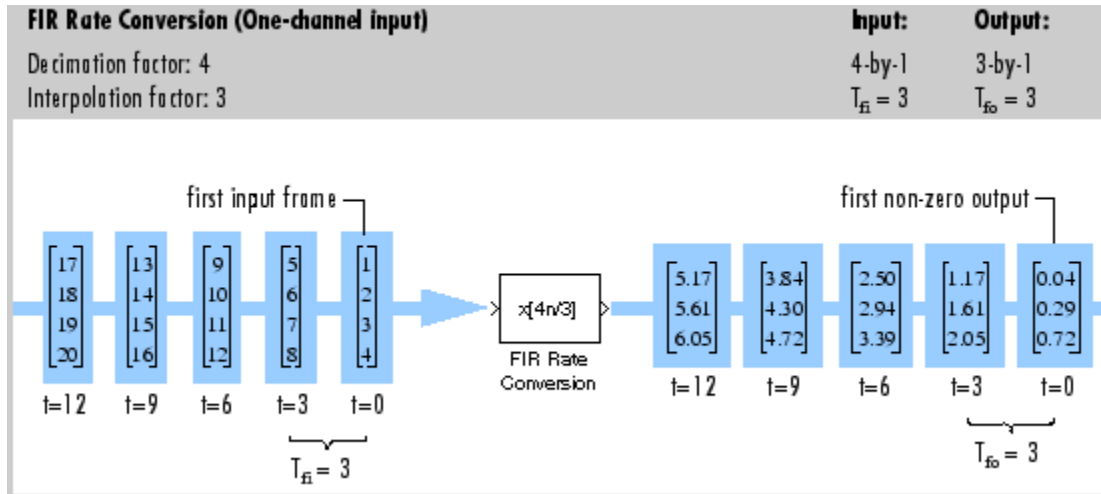
Example 2

The ex_audio_src model provides a simple illustration of one way to convert a speech signal from one sample rate to another. In this model, the data is first sampled at 22,050 Hz and then resampled at 8000 Hz. If you listen to the output, you can hear that the high frequency content has been removed from the signal, although the speech sounds basically the same.

Dialog Box

Coefficient Source

The FIR Rate Conversion block can operate in two different modes. Select the mode in the **Coefficient source** group box. If you select

- **Dialog parameters**, you enter information about the filter such as structure and coefficients in the block mask.

FIR Rate Conversion

- **Multirate filter object (MFILT)**, you specify the filter using an `mfilt` object.

Different items appear on the FIR Rate Conversion block dialog depending on whether you select **Dialog parameters** or **Multirate filter object (MFILT)** in the **Coefficient source** group box. See the following sections for details:

- "Specify Filter Characteristics in Dialog" on page 2-656
- "Specify Multirate Filter Object" on page 2-663

Specify Filter Characteristics in Dialog

The **Main** pane of the FIR Rate Conversion block dialog appears as follows when **Dialog parameters** is selected in the **Coefficient source** group box.
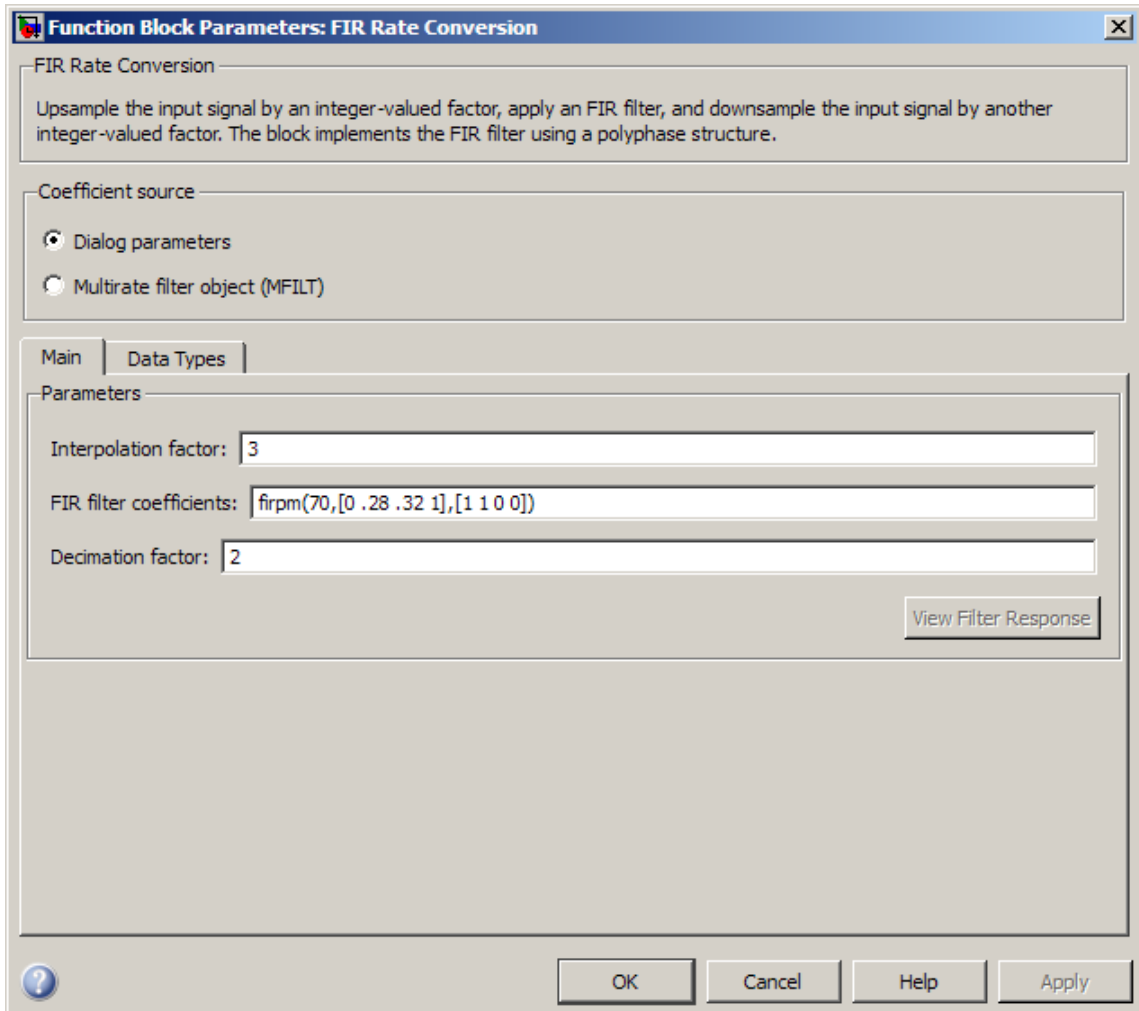
Interpolation factor

Specify the integer factor, L, by which to upsample the signal before filtering.

FIR filter coefficients

Specify the FIR filter coefficients in descending powers of *z*.

Decimation factor

Specify the integer factor, *K*, by which to downsample the signal after filtering.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the filter defined in the block. For more information on FVTool, see the Signal Processing Toolbox documentation.

Note If you specify a filter in the **Multirate filter variable** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

The **Data Types** pane of the FIR Rate Conversion block dialog appears as follows when **Dialog parameters** is specified in the **Coefficient source** group box.

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`.

> **Note** The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when all the following conditions exist:
>
> - **Product output data type** is `Inherit:  Inherit via internal rule`
>
> - **Accumulator data type** is `Inherit:  Inherit via internal rule`
>
> - **Output data type** is `Inherit:  Same as accumulator`
>
> With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See "Fixed-Point Data Types" on page 2-654 and "Multiplication Data Types" for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit:  Same word length as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button $\boxed{\quad \text{>>} \quad}$ to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-654 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-654 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-654 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit:   Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Specify Multirate Filter Object

The **Main** pane of the FIR Rate Conversion block dialog appears as follows when **Multirate filter object (MFILT)** is specified in the **Coefficient source** group box.

Multirate filter variable

Specify the multirate filter object (`mfilt`) that you would like the block to implement. You can do this in one of three ways:

- You can fully specify the `mfilt` object in the block mask.

- You can enter the variable name of a `mfilt` object that is defined in any workspace.

- You can enter a variable name for a `mfilt` object that is not yet defined, as shown in the default value.

For more information on creating `mfilt` objects, see the `mfilt` function reference page.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product and displays the filter response of the `mfilt` object specified in the **Multirate filter variable** parameter. For more information on FVTool, see the Signal Processing Toolbox documentation.
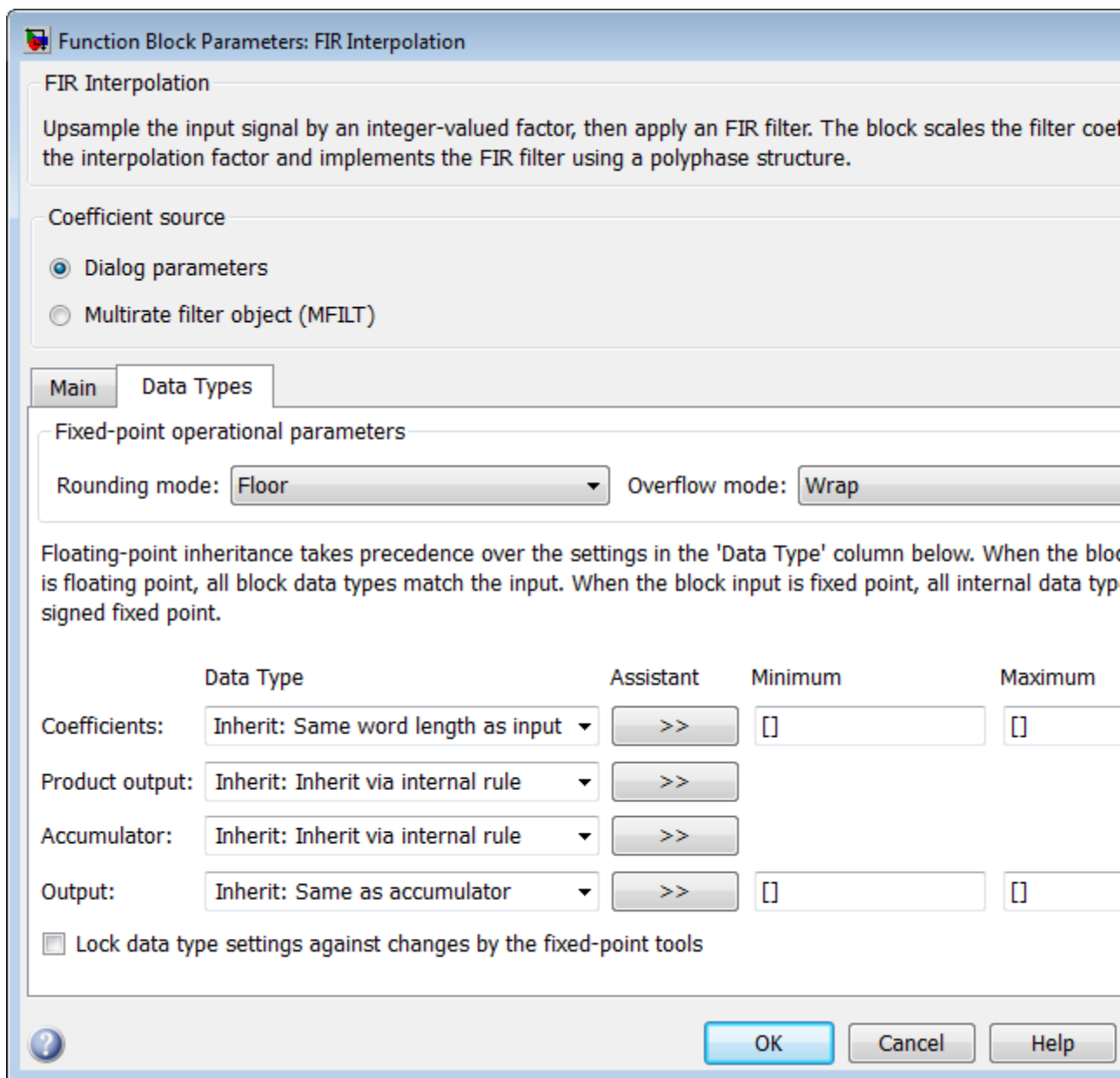

Note If you specify a filter in the **Multirate filter variable** parameter, you must apply the filter by clicking the **Apply** button before using the **View filter response** button.

The **Data Types** pane of the FIR Rate Conversion block dialog appears similar to the following dialog when **Multirate filter object (MFILT)** is specified in the **Coefficient source** group box. The fixed-point data types you see depend on the `mfilt` object you use.

FIR Rate Conversion

The fixed-point settings of the filter object specified on the **Main** pane are displayed on the **Data Types** pane. You cannot change these settings directly on the block mask. To change the fixed-point settings you must edit the filter object directly.

For more information on multirate filter objects, see the `mfilt` function reference page.

References   [1] Orfanidis, S. J. *Introduction to Signal Processing.* Prentice Hall, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Downsample	DSP System Toolbox
FIR Decimation	DSP System Toolbox
FIR Interpolation	DSP System Toolbox
Upsample	DSP System Toolbox
fir1	Signal Processing Toolbox
fir2	Signal Processing Toolbox
firls	Signal Processing Toolbox
upfirdn	Signal Processing Toolbox

FIR Rate Conversion

See the following sections for related information:

- "Convert Sample and Frame Rates in Simulink"
- "Multirate and Multistage Filters"

Purpose   Flip input vertically or horizontally

Library   Signal Management / Indexing

dspindex

Description   The Flip block vertically or horizontally reverses the *M*-by-*N* input
matrix, *u*. The output always has the same dimensionality as the input.

When you set the **Flip along** parameter to Columns, the block flips the
input *vertically* so the first row of the input becomes the last row of
the output.

```
y = flipud(u)    % Equivalent MATLAB code
```

When flipping the input vertically, the block treats length-*M* unoriented
vector inputs as *M*-by-1 column vectors.

When you set the **Flip along** parameter to Rows, the block flips the
input *horizontally* so the first column of the input becomes the last
column of the output.

```
y = fliplr(u)    % Equivalent MATLAB code
```

When flipping the input horizontally, the block treats length-*N*
unoriented vector inputs as 1-by-*N* row vectors.

This block supports Simulink virtual buses.

Flip

Dialog Box
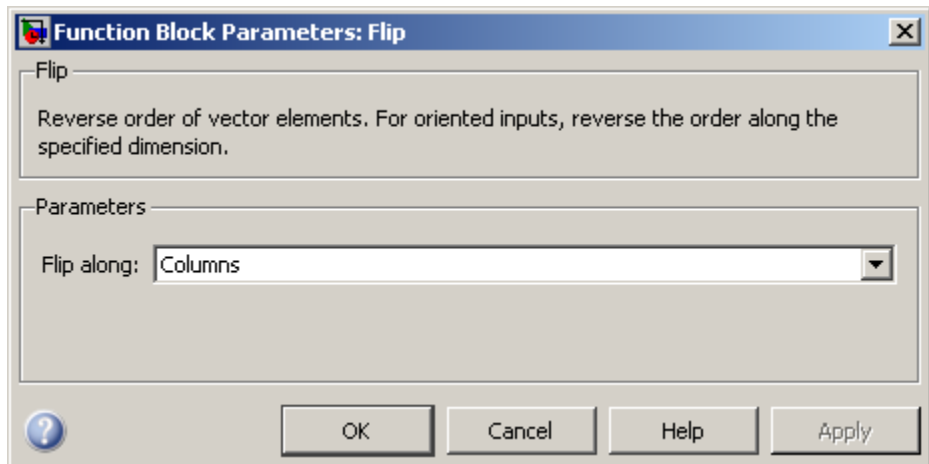
Flip along

    Specify the dimension along which to flip the input. When you set this parameter to Columns, the block flips the input vertically. When you set this parameter to Rows, the block flips the input horizontally.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	• Enumerated

Port	Supported Data Types
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	• Enumerated

See Also

Selector	Simulink
Transpose	DSP System Toolbox
Variable Selector	DSP System Toolbox
flipud	MATLAB
fliplr	MATLAB

Forward Substitution

Purpose   Solve *LX=B* for *X* when *L* is lower triangular matrix

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers

dspsolvers
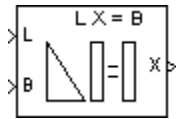
Description   The Forward Substitution block solves the linear system *LX=B* by
simple forward substitution of variables, where:

- *L* is the lower triangular *M*-by-*M* matrix input to the L port.

- *B* is the *M*-by-*N* matrix input to the B port.

The *M*-by-*N* matrix output *X* is the solution of the equations. The block
does not check the rank of the inputs.

The block only uses the elements in the *lower triangle* of input *L* and
ignores the upper elements. When you select **Input L is unit-lower
triangular**, the block assumes the elements on the diagonal of *L* are
1s. 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.

The block treats a length-*M* vector input at port B as an *M*-by-1 matrix.

Fixed-Point Data Types

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

When input L is not unit-lower triangular:
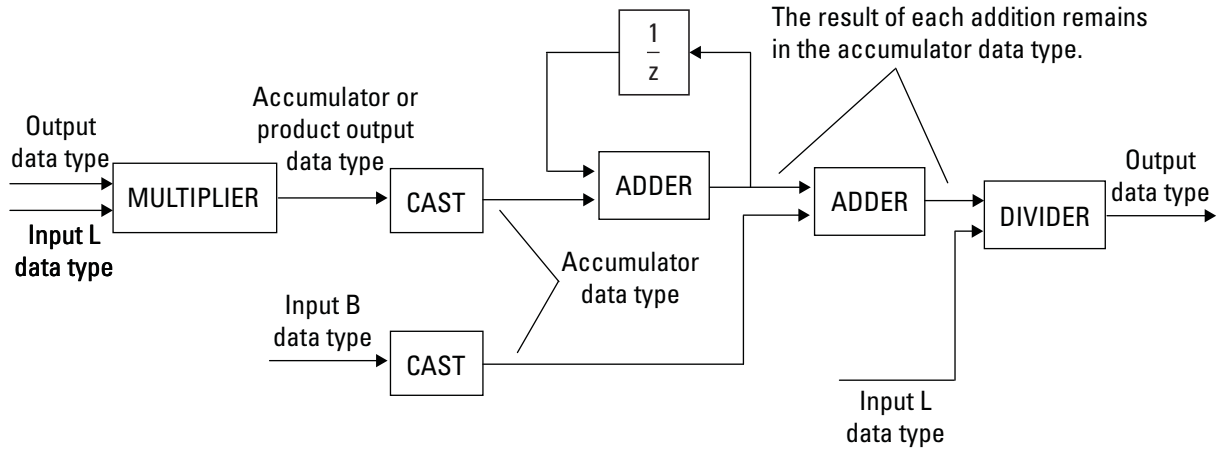
Forward Substitution

When input L is unit-lower triangular:
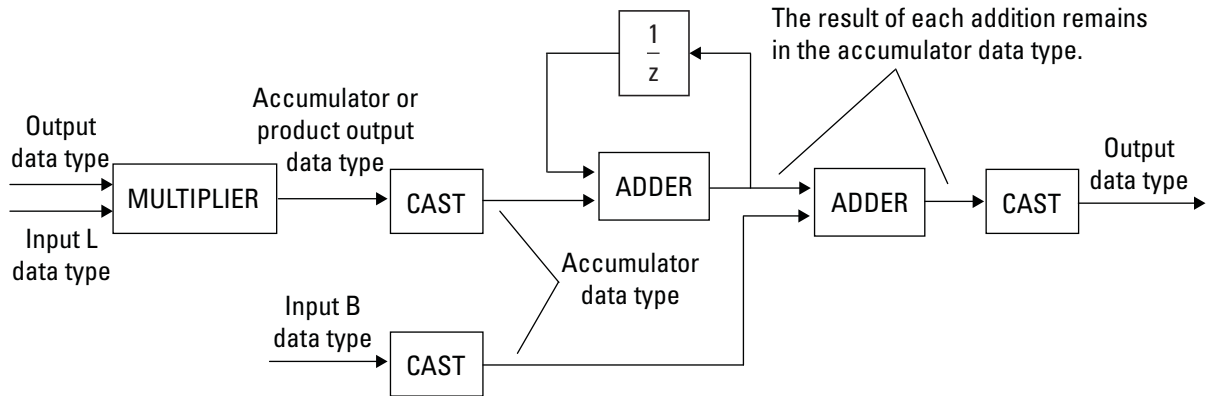
You can set the product output, accumulator, and output data types in the block dialog box, as discussed in the following section.

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

Dialog Box

The **Main** pane of the Forward Substitution block dialog box appears as follows.
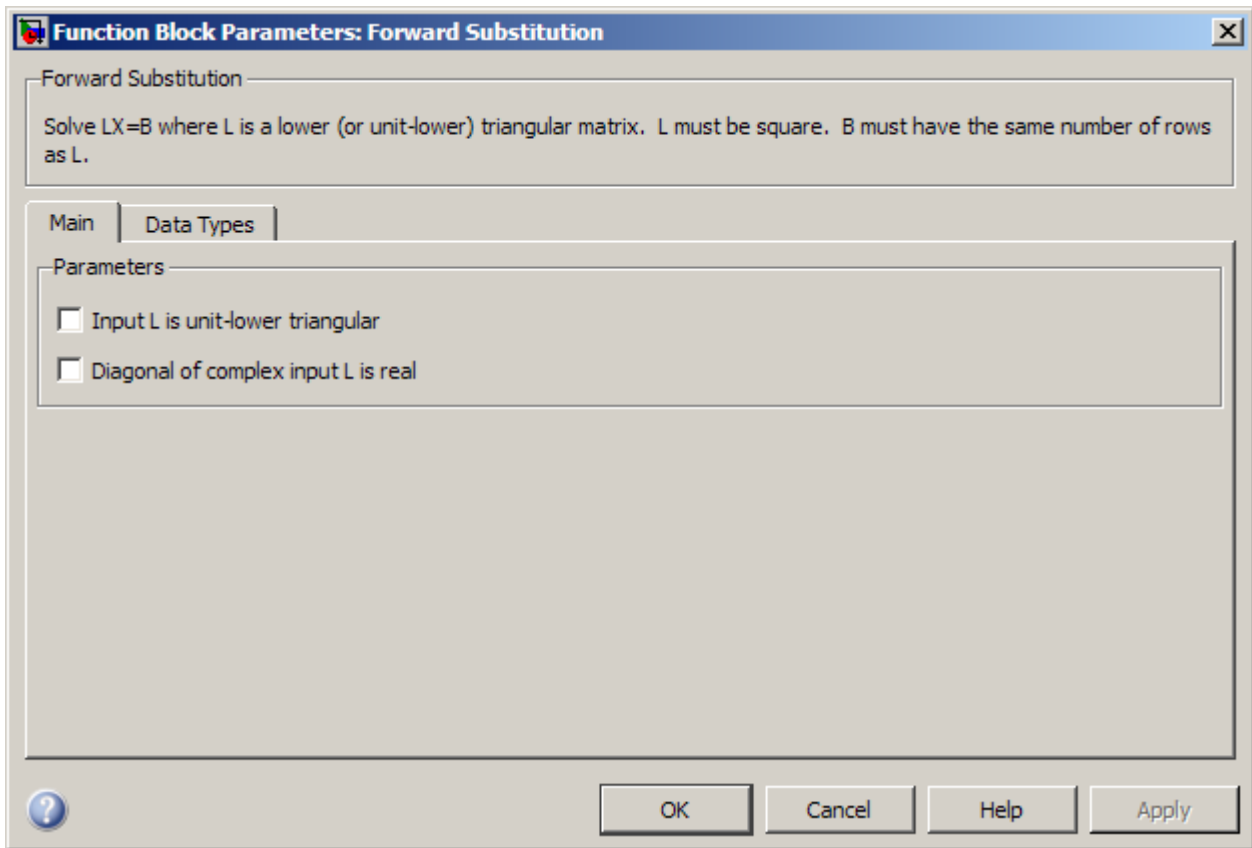
Input L is unit-lower triangular

Select this check box only when all elements on the diagonal of *L* have a value of 1. When you do so, the block optimizes its behavior by skipping an unnecessary divide operation.

Do not select this check box if there are any elements on the diagonal of *L* that do not have a value of 1. When you clear the **Input L is unit-lower triangular** check box, the block always performs the necessary divide operation.

Diagonal of complex input L is real

Select to optimize simulation speed when the diagonal elements of complex input *L* are real. This parameter is only visible when **Input L is unit-upper triangular** is not selected.

Note When L is a complex fixed-point signal, you must select either **Input L is unit-lower triangular** or **Diagonal of complex input L is real**. In such a case, the block ignores any imaginary part of the diagonal of L.

The **Data Types** pane of the Forward Substitution block dialog box appears as follows.
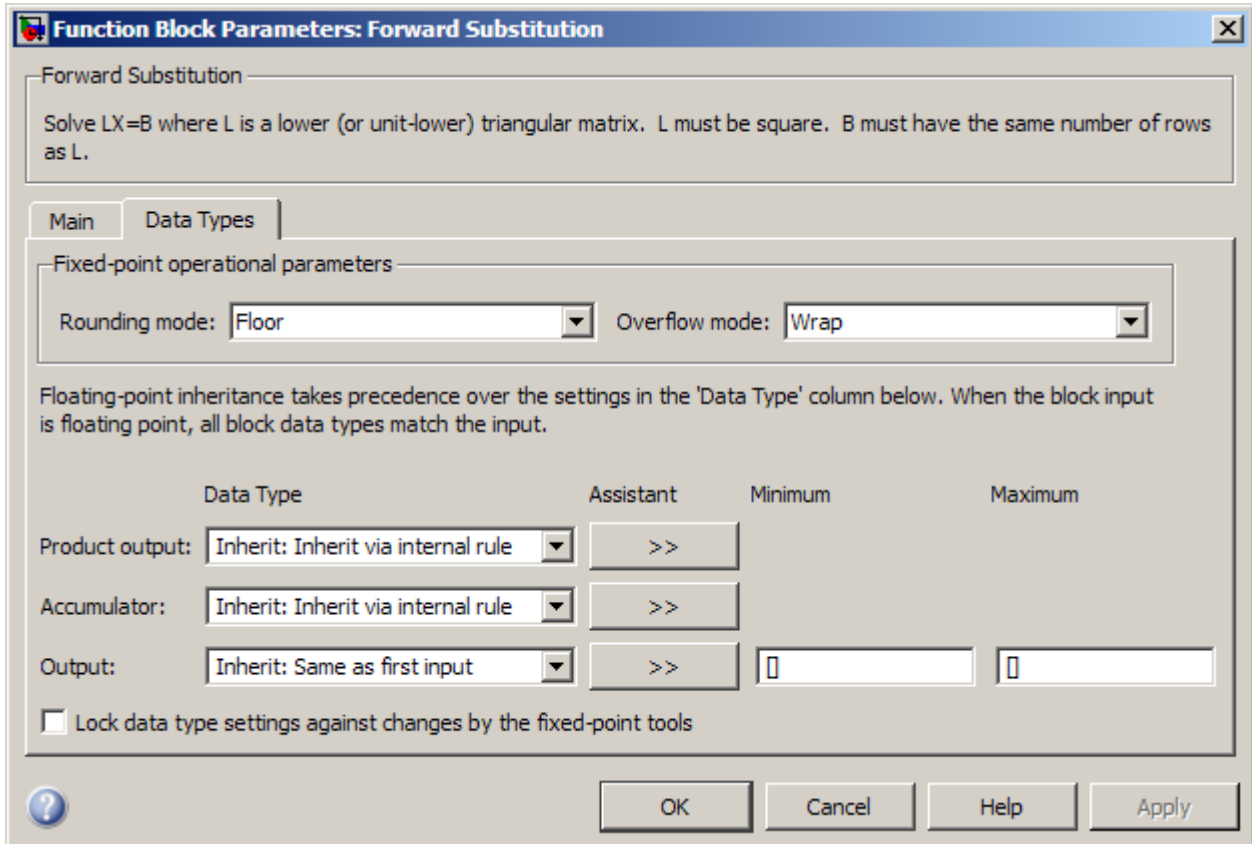
Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-672 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-672 for illustrations depicting the use of the accumulator data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-672 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as first input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Forward Substitution

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
L	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
B	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
X	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

Backward Substitution	DSP System Toolbox
Cholesky Solver	DSP System Toolbox
LDL Solver	DSP System Toolbox
Levinson-Durbin	DSP System Toolbox
LU Solver	DSP System Toolbox
QR Solver	DSP System Toolbox

See "Linear System Solvers" for related information.

Frame Conversion

Purpose　　Specify sampling mode of output signal

Library　　Signal Management / Signal Attributes

　　　　　　dspsigattribs
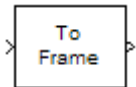
Description　　The Frame Conversion block passes the input through to the output and sets the output sampling mode to the value of the **Sampling mode of output signal** parameter, which can be either Frame-based or Sample-based. The output sampling mode can also be inherited from the signal at the Ref (reference) input port, which you make visible by selecting the **Inherit output sampling mode from <Ref> input port** check box.

The Frame Conversion block does not make any changes to the input signal other than the sampling mode. In particular, the block does not rebuffer or resize 2-D inputs. Because 1-D vectors cannot be frame based, when the input is a length-M 1-D vector and the block is in Frame-based mode, the output is a frame-based M-by-1 matrix — that is, a single channel.
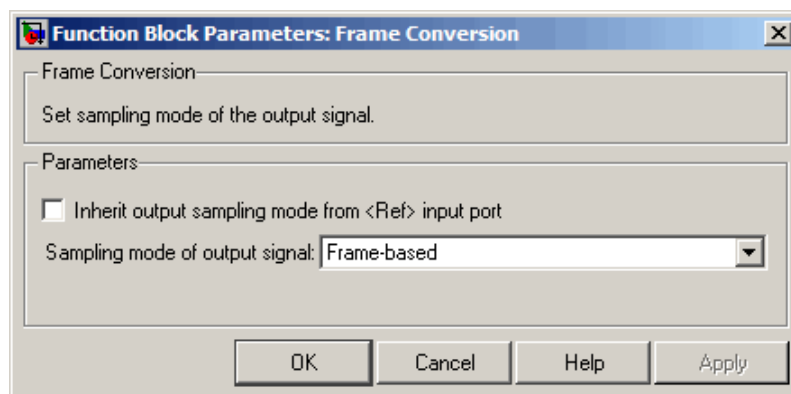
Dialog Box

Inherit output sampling mode from <Ref> input port
　　Select to enable the Ref port from which the block inherits the output sampling mode.

Sampling mode of output signal
> Specify the sampling mode of the output signal, Frame-based or Sample-based.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Ref	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated

Frame Conversion

See Also

Buffer	DSP System Toolbox
Check Signal Attributes	DSP System Toolbox
Convert 1-D to 2-D	DSP System Toolbox
Convert 2-D to 1-D	DSP System Toolbox
Inherit Complexity	DSP System Toolbox
Unbuffer	DSP System Toolbox
Probe	Simulink
Reshape	Simulink
Signal Specification	Simulink

Purpose   Specify frame status of output as sample based or frame based

Library   dspobslib

Description

> Frame-
> based

> In   Inherit
>   Frame
> Ref Status
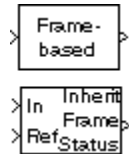
Note The Frame Status Conversion block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Frame Conversion block.

The Frame Status Conversion block passes the input through to the output, and sets the output frame status to the **Output signal** parameter, which can be either Frame-based or Sample-based. The output frame status can also be inherited from the signal at the Ref (reference) input port, which is made visible by selecting the **Inherit output frame status from Ref input port** check box.

When the **Output signal** parameter setting or the inherited signal's frame status differs from the input frame status, the block changes the input frame status accordingly, but does not otherwise alter the signal. In particular, the block does not rebuffer or resize 2-D inputs. Because 1-D vectors cannot be frame based, when the input is a length-M 1-D vector, and the **Output signal** parameter is set to Frame-based, the output is a frame-based M-by-1 matrix (that is, a single channel).

When the **Output signal** parameter or the inherited signal's frame status matches the input frame status, the block passes the input through to the output unaltered.

Frame Status Conversion (Obsolete)

Dialog Box
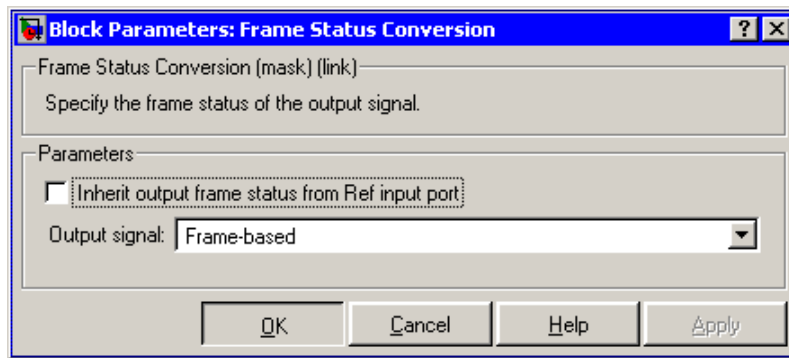
Inherit output frame status from Ref input port

> When selected, enables the Ref input port from which the block inherits the output frame status.

Output signal

> The output frame status, Frame-based or Sample-based.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Ref	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• Boolean

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Check Signal Attributes	DSP System Toolbox
Convert 1-D to 2-D	DSP System Toolbox
Convert 2-D to 1-D	DSP System Toolbox
Inherit Complexity	DSP System Toolbox

From Audio Device

Purpose	Read audio data from computer's audio device
Library	Sources
	dspsrcs4

Description

The From Audio Device block reads audio data from an audio device in real time. This block is not supported for use with the Simulink Model block.

Use the **Device** parameter to specify the device from which to acquire audio. This parameter is automatically populated based on the audio devices installed on your system. If you plug or unplug an audio device from your system, type clear mex at the MATLAB command prompt to update this list.

Use the **Number of channels** parameter to specify the number of audio channels in the signal. For example:

- Enter 2 if the audio source is two channels (stereo).
- Enter 1 if the audio source is single channel (mono).
- Enter 6 if you are working with a 5.1 speaker system.

The block's output is an *M*-by-*N* matrix, where *M* is the number of consecutive samples and *N* is the number of audio channels.

Use the **Sample rate (Hz)** parameter to specify the number of samples per second in the signal. If the audio data is processed in uncompressed pulse code modulation (PCM) format, it should typically be sampled at one of the standard audio device rates: 8000, 11025, 22050, 44100, or 48000 Hz.

The range of supported audio device sample rates and data type formats, depend on both the sound card and the API which is chosen for the sound card.

Use the **Device data type** parameter to specify the data type of the audio data that the device is placing in the buffer. You can choose:

- 8-bit integer

- 16-bit integer

- 24-bit integer

- 32-bit float

- Determine from output data type

If you choose `Determine from output data type`, the following table summarizes the block's behavior.

Output Data Type	Device Data Type
Double-precision floating point or single-precision floating point	32-bit floating point
32-bit integer	24-bit integer
16-bit integer	16-bit integer
8-bit integer	8-bit integer

If you choose `Determine from output data type` and the device does not support a data type, the block uses the next lowest precision data type supported by the device.

Use the **Frame size (samples)** parameter to specify the number of samples in the block's output. Use the **Output data type** parameter to specify the data type of audio data output by the block.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra .dll files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

Buffering

The From Audio Device block buffers the data from the audio device using the process illustrated by the following figure.

From Audio Device
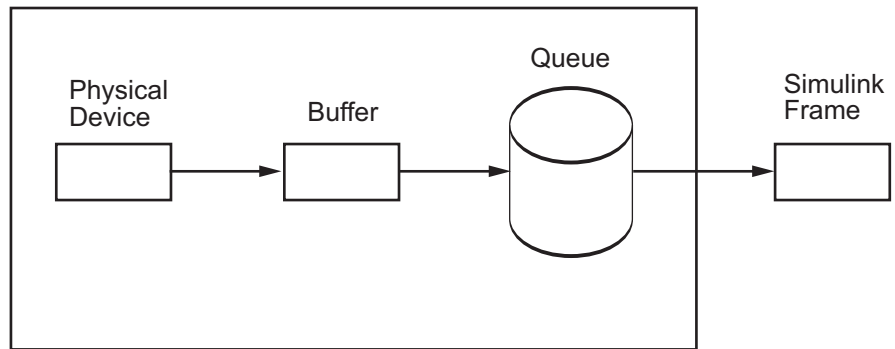
From Audio Device Block

1 At the start of the simulation, the audio device begins writing the input data to a buffer. This data has the data type specified by the **Device data type** parameter.

2 When the buffer is full, the From Audio Device block writes the contents of the buffer to the queue. Specify the size of this queue using the **Queue duration (seconds)** parameter.

3 As the audio device appends audio data to the bottom of the queue, the From Audio Device block pulls data from the top of the queue to fill the Simulink frame. This data has the data type specified by the **Output data type** parameter.

Select the **Automatically determine buffer size** check box to allow the block to calculate a conservative buffer size using the following equation:

$$size = 2^{\left\lfloor \log_2 \frac{sr}{10} \right\rfloor}$$

In this equation, *size* is the buffer size, and *sr* is the sample rate. If you clear this check box, the **Buffer size (samples)** parameter appears on the block. Use this parameter to specify the buffer size in samples.

When the simulation throughput rate is lower than the hardware throughput rate, the queue, which is initially empty, fills up. If the queue is full, the block drops the incoming data from the audio device. When the simulation throughput rate is higher than the hardware throughput rate, the From Audio Device block waits for new samples to become available.

Troubleshooting

When Simulink cannot keep up with an audio device that is operating in real time, the queue fills up and the block begins to lose audio data. Here are several ways to deal with this situation:

- *Increase the queue duration.*

  The **Queue duration (seconds)** parameter specifies the duration of the signal, in seconds, that can be buffered during the simulation. This is the maximum length of time that the block's data demand can lag behind the hardware's data supply.

- *Increase the buffer size.*

  The size of the buffer processed in each interrupt from the audio device affects the performance of your model. If the buffer is too small, a large portion of hardware resources are used to write data to the queue. If the buffer is too big, Simulink must wait for the device to fill the buffer before it moves the data to the queue, which introduces 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 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 Simulink Coder code generation software. Native code runs much faster than Simulink and should provide rates adequate for real-time audio processing.

Other ways to improve throughput rates include simplifying the model and running the simulation on a faster PC processor. For other ideas on improving simulation performance, see "Delay and Latency" and "Performance" in the Simulink documentation.

Audio Hardware API

The To Audio Device and From Audio Device blocks use the open-source PortAudio library in order to communicate with the audio hardware on a given computer. The PortAudio library supports a range of API's designed to communicate with the audio hardware on a given platform. The following API choices were made when building the PortAudio library for the DSP System Toolbox product:

- Windows: DirectSound, WDM—KS, ASIO

- Linux: OSS, ALSA

- Mac: CoreAudio

To select or change the Audio Hardware API, select **Preferences** from the MATLAB Toolstrip. Then select DSP System Toolbox from the tree menu. In the **DSP System Toolbox Preferences** dialog box, the option is disabled if no device corresponds to that particular audio API.

Dialog Box

Source Block Parameters: From Audio Device

From Audio Device

Record sound data from your computer's audio device.

Parameters

Device: Default

Number of channels: 2

Sample rate (Hz): 44100

Device data type: Determine from output data type

☑ Automatically determine buffer size

Queue duration (seconds): 1.0

Outputs

Frame size (samples): 1024

Output data type: single

[OK]   [Cancel]   [Help]

Device
Specify the device from which to acquire audio data.

Number of channels
Specify the number of audio channels.

Sample rate (Hz)
Specify the number of samples per second in the signal.

Device data type
Specify the data type used by the device to acquire audio data.

From Audio Device

Automatically determine buffer size

Select this check box to enable the block to use a conservative buffer size.

Buffer size (samples)

Specify the size of the buffer that the block uses to communicate with the audio device. This parameter is visible when the **Automatically determine buffer size** check box is cleared.

Queue duration (seconds)

Specify the size of the queue in seconds.

Frame size (samples)

Specify the number of samples in the block's output signal.

Output data type

Select the data type of the block's output.

Supported Data Types

Port	Supported Data Types
Output	• Double-precision floating point
	• Single-precision floating point
	• 32-bit signed integers
	• 16-bit signed integers
	• 8-bit unsigned integers

See Also

From Multimedia File	DSP System Toolbox
To Audio Device	DSP System Toolbox
audiorecorder	MATLAB
dsp.AudioRecorder	System Object

Purpose   Read multimedia file

Library   Sources

dspsrcs4

From Multimedia File1
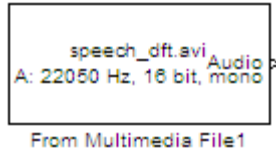
Description

The From Multimedia File block reads audio samples, video frames, or both from a multimedia file. The block imports data from the file into a Simulink model.

Note This block supports code generation for the host computer that has file I/O available. You cannot use this block with Real-Time Windows Target™ software because that product does not support file I/O.

The generated code for this block relies on prebuilt library files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra library files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

To run an executable file that was generated from a model containing this block, you may need to add precompiled shared library files to your system path. See "Understanding Code Generation" for details.

From Multimedia File

Supported Platforms and File Types

The supported file formats available to you depends on the codecs installed on your system.

Windows® Platforms Supported File Formats

With the necessary Windows DirectShow codecs installed on your system, the From Multimedia File Block supports many video and audio file formats. This block performs best on platforms with Version 9.0 or later of DirectX® software.

The following table lists the most common file formats.

Multimedia Types	File Name Extensions
Image files	`.jpg,.bmp`
Video files	`.qt, .mov, .avi, .asf, .asx, .wmv, .mpg, .mpeg, .mp2, .mp4, .m4v`
Audio files	`.wav, .wma, .aif, .aifc, .aiff, .mp3, .au, .snd, .mp4, .m4a, .flac, .ogg`

Windows XP x64 platform ships with a limited set of 64-bit video and audio codecs. If the From Multimedia File block cannot work on a compressed multimedia file, try one of the two alternatives:

- Run the 32-bit version of MATLAB on your Windows XP x64 platform. Windows XP x64 ships with many 32-bit codecs.

- Save the multimedia file to a file format supported by the From Multimedia File block.

If you use Windows, use Windows Media player Version 11 or later with this block for best results.

Non-Windows Platform Supported File Formats

The following table lists the most common file formats.

Multimedia Types	File Name Extensions
Video files	`.avi, .mj2, .mov, .mp4,.m4v`
Audio files	Uncompressed `.avi, .mp3, .mp4, .m4a, .wav, .flac, .ogg`

Ports

The output ports of the From Multimedia File block change according to the content of the multimedia file. If the file contains only video frames, the **Image**, intensity **I**, or **R**,**G**,**B** ports appear on the block. If the file contains only audio samples, the **Audio** port appears on the block. If the file contains both audio and video, you can select the data to emit. The following table describes available ports.

Port	Description
Image	M-by-N-by-P color video signal where P is the number of color planes.
I	M-by-N matrix of intensity values.
R, **G**, **B**	Matrix that represents one plane of the RGB video stream. Outputs from the R, G, or B ports must have same dimensions.
Audio	Vector of audio data.
Y, Cb, Cr	Matrix that represents one frame of the YCbCr video stream. The Y, Cb, Cr ports produce the following outputs: Y: $M \ x \ N$ Cb: $M \ x \dfrac{N}{2}$ Cr: $M \ x \dfrac{N}{2}$

Sample Rates

The sample rate that the block uses depends on the audio and video sample rate. While the FMMF block operates at a single rate in Simulink, the underlying audio and video streams can produce different

From Multimedia File

rates. In some cases, when the block outputs both audio and video, makes a small adjustment to the video rate.

Sample Time Calculations Used for Video and Audio Files

Sample time = $\dfrac{ceil(AudioSampleRate/FPS)}{AudioSampleRate}$.

When audio sample time, $\dfrac{AudioSampleRate}{FPS}$ is noninteger, the equation cannot reduce to $\dfrac{1}{FPS}$.

In this case, to prevent synchronization problems, the block drops the corresponding video frame when the audio stream leads the video stream by more than $\dfrac{1}{FPS}$.

In summary, the block outputs one video frame at each Simulink time step. To calculate the number of audio samples to output at each time step, the block divides the audio sample rate by the video frame rate (fps). If the audio sample rate does not divide evenly by the number of video frames per second, the block rounds the number of audio samples up to the nearest whole number. If necessary, the block periodically drops a video frame to maintain synchronization for large files.

Dialog Box

The **Main** pane of the From Multimedia File block dialog appears as follows.
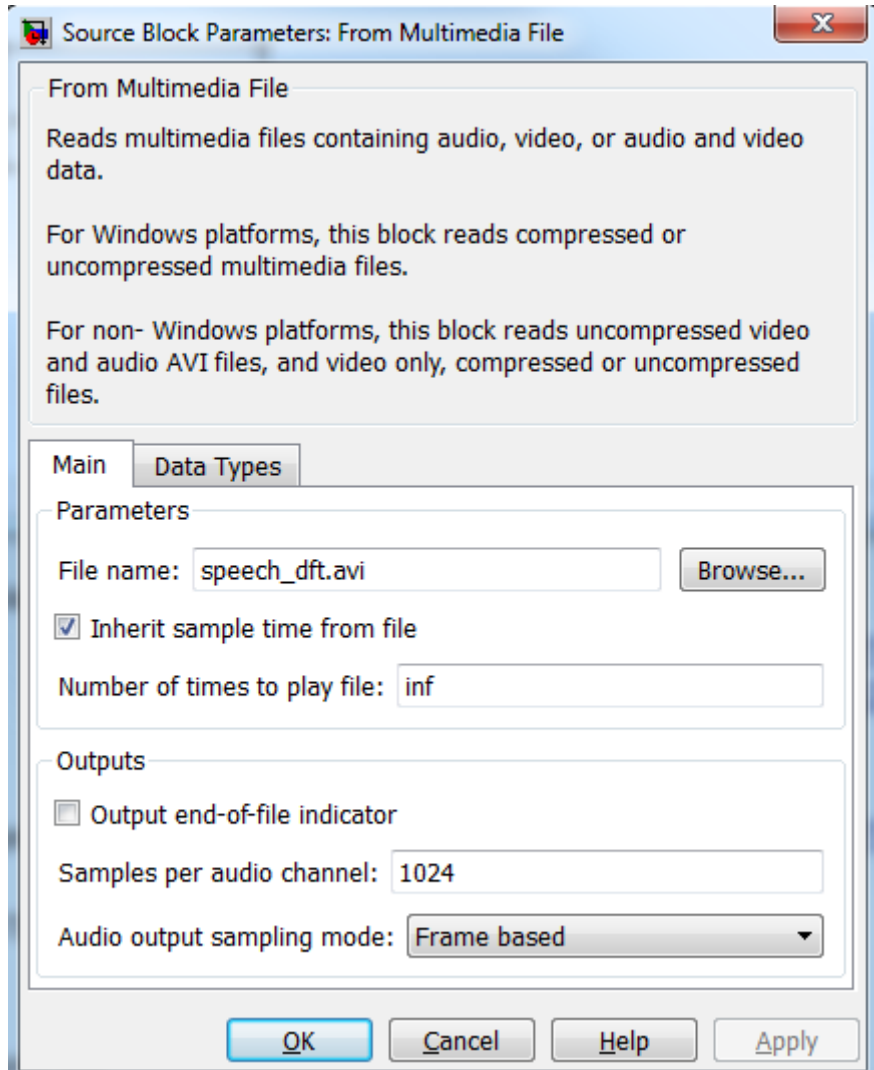
From Multimedia File

File name
>Specify the name of the multimedia file from which to read. The block determines the type of file (audio and video, audio only, or video only) and provides the associated parameters.
>
>If the location of the file does not appear on your MATLAB path, use the **Browse** button to specify the full path. Otherwise, if the location of this file appears on your MATLAB path, enter only the file name. On Windows platforms, this parameter supports URLs that point to MMS (Microsoft Media Server) streams.

Inherit sample time from file
>Select the **Inherit sample time from file** check box if you want the block sample time to be the same as the multimedia file. If you clear this check box, enter the block sample time in the **Desired sample time** parameter field. The file that the From Multimedia File block references, determines the block default sample time. You can also set the sample time for this block manually. If you do not know the intended sample rate of the video, let the block inherit the sample rate from the multimedia file.

Desired sample time
>Specify the block sample time. This parameter becomes available if you clear the **Inherit sample time from file** check box.

Number of times to play file
>Enter a positive integer or `inf` to represent the number of times to play the file.

Output end-of-file indicator
>Use this check box to determine whether the output is the last video frame or audio sample in the multimedia file. When you select this check box, a Boolean output port labeled EOF appears on the block. The output from the EOF port defaults to 1 when the last video frame or audio sample is output from the block. Otherwise, the output from the EOF port defaults to 0.

Multimedia outputs

Specify `Video and audio`, `Video only`, or `Audio only` output file type. This parameter becomes available only when a video signal has both audio and video.

Samples per audio channel

Specify number of samples per audio channel. This parameter becomes available for files containing audio.

Output color format

Specify whether you want the block to output `RGB`, `Intensity`, or `YCbCr 4:2:2` video frames. This parameter becomes available only for a signal that contains video. If you select `RGB`, use the **Image signal** parameter to specify how to output a color signal.

Image signal

Specify how to output a color video signal. If you select `One multidimensional signal`, the block outputs an M-by-N-by-P color video signal, where P is the number of color planes, at one port. If you select `Separate color signals`, additional ports appear on the block. Each port outputs one M-by-N plane of an RGB video stream. This parameter becomes available only if you set the **Image color space** parameter to `RGB` and the signal contains video.

Audio output sampling mode

Select Sample based or Frame based output. If the input to the block has 3 or more dimensions, you must select Sample based output. This parameter appears when you specify a file containing audio for the **File name** parameter.

The **Data Types** pane of the To Multimedia File block dialog box appears as follows.

From Multimedia File
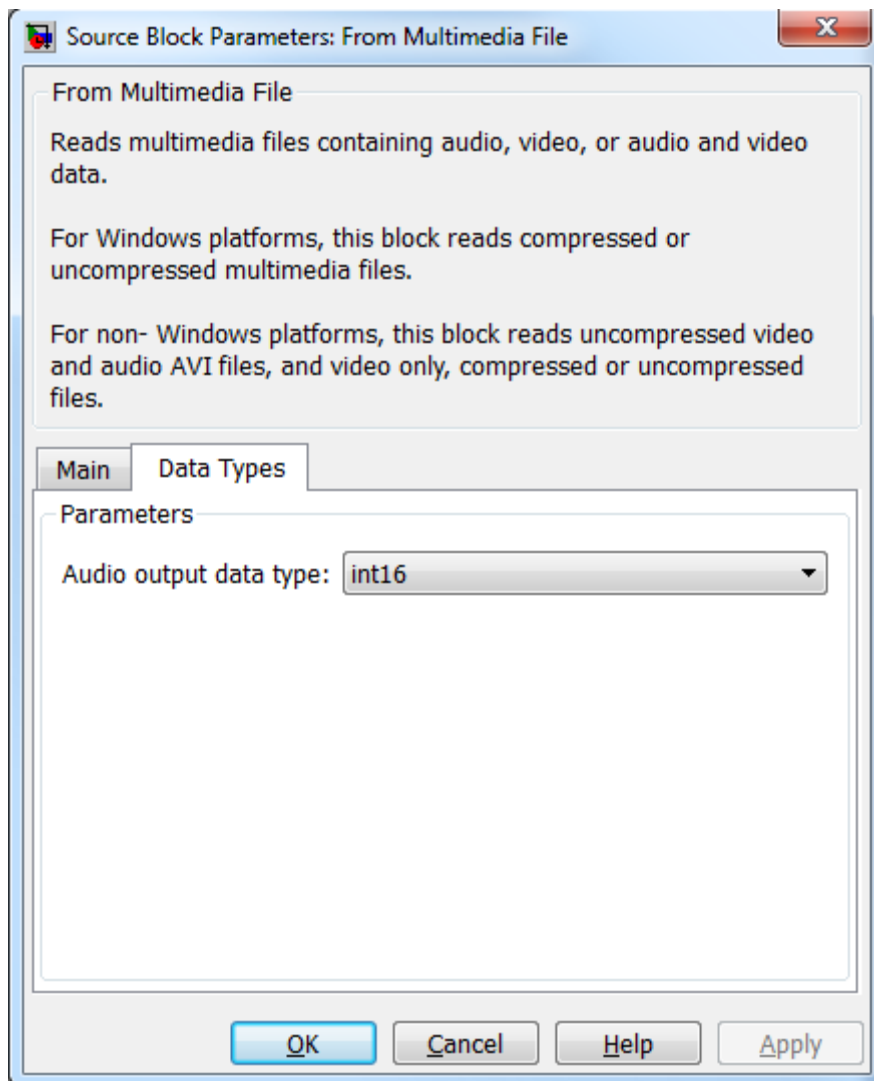
Audio output data type

Set the data type of the audio samples output at the Audio port. This parameter becomes available only if the multimedia file contains audio. You can choose double, single, int16, or uint8 types.

Video output data type

Set the data type of the video frames output at the **R**, **G**, **B**, or **Image** ports. This parameter becomes available only if the multimedia file contains video. You can choose double, single, int8, uint8, int16, uint16, int32, uint32, or Inherit from file types.

Supported Data Types

For source blocks to display video data properly, double- and single-precision floating-point pixel values must be between 0 and 1. For other data types, the pixel values must be between the minimum and maximum values supported by their data type.

Port	Supported Data Types	Supports Complex Values?
Image	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers	No
R, G, B	Same as the Image port	No
Audio	• Double-precision floating point • Single-precision floating point • 16-bit signed integers • 8-bit unsigned integers	No
Y, Cb,Cr	Same as the Image port	No

From Multimedia File

To Multimedia File	DSP System Toolbox
"Sample Time"	Simulink

Purpose   Read audio data from standard audio device in real-time (32-bit Windows operating systems only)

Library   dspwin32

Description

Note The From Wave Device block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the From Audio Device block.

The From Wave Device block reads audio data from a standard Windows audio device in real-time. It is compatible with most popular Windows hardware, including Sound Blaster cards. (Models that contain both this block and the To Wave Device block require a *duplex-capable* sound card.)

The **Use default audio device** parameter allows the block to detect and use the system's default audio hardware. This option should be selected on systems that have a single sound device installed, or when the default sound device on a multiple-device system is the desired source. In cases when the default sound device is not the desired input source, clear **Use default audio device**, and select the desired device in the **Audio device menu** parameter.

When the audio source contains two channels (stereo), the **Stereo** check box should be selected. When the audio source contains a single channel (mono), the **Stereo** check box should be cleared. For stereo input, the block's output is an M-by-2 matrix containing one frame (M consecutive samples) of audio data from each of the two channels. For mono input, the block's output is an M-by-1 matrix containing one frame (M consecutive samples) of audio data from the mono input. The frame size, M, is specified by the **Samples per frame** parameter. For M=1, the output is sample based; otherwise, the output is frame based.

The audio data is processed in uncompressed pulse code modulation (PCM) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. You can

From Wave Device (Obsolete)

select one of these rates from the **Sample rate** parameter. To specify a different rate, select the **User-defined** option and enter a value in the **User-defined sample rate** parameter.

The **Sample Width (bits)** parameter specifies the number of bits used to represent the signal samples read by the audio device. The following settings are available:

- 8 — allocates 8 bits to each sample, allowing a resolution of 256 levels

- 16 — allocates 16 bits to each sample, allowing a resolution of 65536 levels

- 24 — allocates 24 bits to each sample, allowing a resolution of 16777216 levels (only for use with 24-bit audio devices)

Higher sample width settings require more memory but yield better fidelity. The output from the block is independent of the **Sample width (bits)** setting. The output data type is determined by the **Data type** parameter setting.

Buffering

Since the audio device accepts real-time audio input, Simulink software must read a continuous stream of data from the device throughout the simulation. Delays in reading data from the audio hardware can result in hardware errors or distortion of the signal. This means that the From Wave Device block must read data from the audio hardware as quickly as the hardware itself acquires the signal. However, the block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running from within Simulink rather than as generated code. (Simulink operations are generally slower than comparable hardware operations, and execution speed routinely varies during the simulation as the host operating system services other processes.) The block must therefore rely on a buffering strategy to ensure that signal data can be read on schedule without losing samples.

At the start of the simulation, the audio device begins writing the input data to a (hardware) buffer with a capacity of T_b seconds. The From Wave Device block immediately begins pulling the earliest samples off

the buffer (first in, first out) and collecting them in length-M frames for output. As the audio device continues to append inputs to the bottom of the buffer, the From Wave Device block continues to pull inputs off the top of the buffer at the best possible rate.

The following figure shows an audio signal being acquired and output with a frame size of 8 samples. The buffer of the sound board is approaching its five-frame capacity at the instant shown, which means that the hardware is adding samples to the buffer more rapidly than the block is pulling them off. (If the signal sample rate was 8 kHz, this small buffer could hold approximately 0.005 second of data.
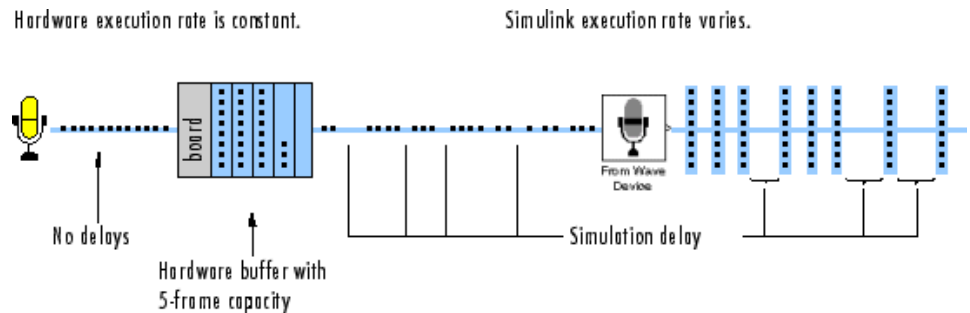
When the simulation throughput rate is higher than the hardware throughput rate, the buffer remains empty throughout the simulation. If necessary, the From Wave Device block simply waits for new samples to become available on the buffer (the block does not interpolate between samples). More typically, the simulation throughput rate is lower than the hardware throughput rate, and the buffer tends to fill over the duration of the simulation.

Troubleshooting

When the buffer size is too small in relation to the simulation throughput rate, the buffer might fill before the entire length of signal is processed. This usually results in a device error or undesired device output. When this problem occurs, you can choose to either increase the buffer size or the simulation throughput rate:

- *Increase the buffer size*

  The **Queue duration** parameter specifies the duration of signal, T_b (in real-time seconds), that can be buffered in hardware during the simulation. Equivalently, this is the maximum length of time that the block's data acquisition can lag the hardware's data acquisition. The number of frames buffered is approximately

  $$\frac{T_b F_s}{M}$$

  where F_s is the sample rate of the signal and M is the number of samples per frame. The required buffer size for a given signal depends on the signal length, the frame size, and the speed of the simulation. Note that increasing the buffer size might increase model latency.

- *Increase the simulation throughput rate*

  Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code:

  - Increase frame sizes (and convert sample-based signals to frame-based signals) throughout the model to reduce the amount of block-to-block communication overhead. This can drastically increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations.

  - Generate executable code with Simulink Coder. 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" and "Performance" in the Simulink documentation for other ideas on improving simulation performance.

**Dialog
Box**
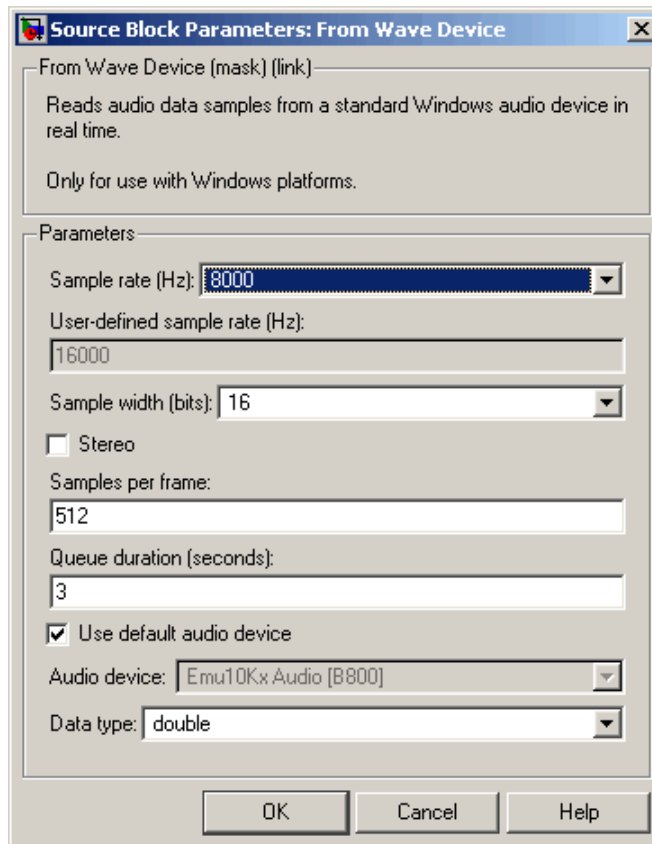
Sample rate (Hz)

    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.

From Wave Device (Obsolete)

Stereo

Specifies stereo (two-channel) inputs when selected, mono (one-channel) inputs when cleared. Stereo output is *M*-by-2; mono output is *M*-by-1.

Samples per frame

The number of audio samples in each successive output frame, *M*. When the value of this parameter is 1, the block outputs a sample-based signal.

Queue duration (seconds)

The length of signal (in seconds) to buffer to the hardware at the start of the simulation.

Use default audio device

Reads audio input from the system's default audio device when selected. Clear to enable the **Audio device ID** parameter and select a device.

Audio device

The name of the audio device from which to read the audio output (lists the names of the installed audio device drivers). Select **Use default audio device** when the system has only a single audio card installed.

Data type

The data type of the output: double-precision, single-precision, signed 16-bit integer, or unsigned 8-bit integer.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

See Also

From Wave File (Obsolete)	DSP System Toolbox
To Wave Device (Obsolete)	DSP System Toolbox
audiorecorder	MATLAB

From Wave File (Obsolete)

Purpose   Read audio data from Microsoft Wave (.wav) file

Library   dspwin32

Description
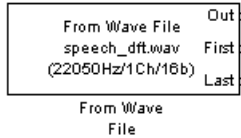
Note The From Wave File block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the From Multimedia File block.

The From Wave File block streams audio data from a Microsoft® Wave (.wav) file and generates a signal with one of the data types and amplitude ranges in the following table.

Note AVI files are the only supported file type for non-Windows platforms.

Output Data Type	Output Amplitude Range
double	±1
single	±1
int16	-32768 to 32767 (-2^{15} to 2^{15} - 1)
uint8	0 to 255

The audio data must be in uncompressed pulse code modulation (PCM) format.

```
y = wavread('filename')      % Equivalent MATLAB code
```

The block supports 8-, 16-, 24-, and 32-bit Microsoft Wave (.wav) files.

The **File name** parameter can specify an absolute or relative path to the file. When the file is on the MATLAB path or in the current folder (the

folder returned by typing `pwd` at the MATLAB command line), you need only specify the file name. You do not need to specify the `.wav` extension.

Note The From Wave File block does not support .wav file names that contain a + character. To read .wav files that have a + character in the file name, use the From Multimedia File block.

For an audio file containing C channels, the block's output is an M-by-C matrix containing one frame (M consecutive samples) of audio data from each channel. The frame size, M, is specified by the **Samples per output frame** parameter. For M=1, the output is sample based; otherwise, the output is frame based.

The output frame period, T_{fo}, is

$$T_{fo} = \frac{M}{F_s}$$

where F_s is the data sample rate in Hz.

To reduce the required number of file accesses, the block acquires L consecutive samples from the file during each access, where L is specified by the **Minimum number of samples for each read from file** parameter ($L \geq M$). For $L < M$, the block instead acquires M consecutive samples during each access. Larger values of L result in fewer file accesses, which reduces run-time overhead.

Use the **Data type** parameter to specify the data type of the block's output. Your choices are `double`, `single`, `uint8`, or `int16`.

Select the **Loop** check box if you want to play the file more than once. Then, enter the number of times to play the file. The number you enter must be a positive integer or `inf`.

Use the **Number of times to play file** parameter to enter the number of times to play the file. The number you enter must be a positive integer or `inf`, to play the file until you stop the simulation.

From Wave File (Obsolete)

The **Samples restart** parameter determines whether the samples from the audio file repeat immediately or repeat at the beginning of the next frame output from the output port. When you select `immediately after last sample`, the samples repeat immediately. When you select `at beginning of next frame`, the frame containing the last sample value from the audio file is zero padded until the frame is filled. The block then places the first sample of the audio file in the first position of the next output frame.

Use the **Output start-of-file indicator** parameter to determine when the first audio sample in the file is output from the block. When you select this check box, a Boolean output port labeled SOF appears on the block. The output from the SOF port is 1 when the first audio sample in the file is output from the block. Otherwise, the output from the SOF port is 0.

Use the **Output end-of-file indicator** parameter to determine when the last audio sample in the file is output from the block. When you select this check box, a Boolean output port labeled EOF appears on the block. The output from the EOF port is 1 when the last audio sample in the file is output from the block. Otherwise, the output from the EOF port is 0.

The block icon shows the name, sample rate (in Hz), number of channels (1 or 2), and sample width (in bits) of the data in the specified audio file. All sample rates are supported; the sample width must be either 8, 16, 24, or 32 bits.

Dialog Box
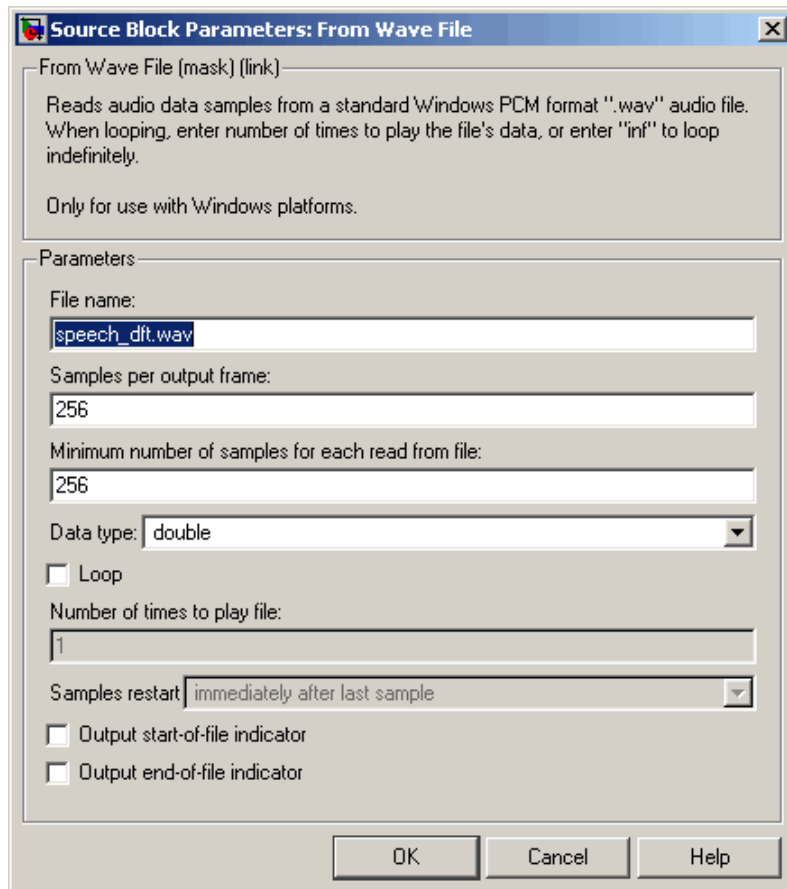
File name

> Enter the path and name of the file to read. Paths can be relative or absolute.

From Wave File (Obsolete)

> **Note** The From Wave File block does not support .wav file names that contain a + character. To read .wav files that have a + character in the file name, use the From Multimedia File block.

Samples per output frame
> Enter the number of samples in each output frame, *M*. When the value of this parameter is 1, the block outputs a sample-based signal.

Minimum number of samples for each read from file
> Enter the number of consecutive samples to acquire from the file with each file access, *L*.

Data type
> Select the output data type: `double`, `single`, `uint8`, or `int16`. The data type setting determines the output's amplitude range, as shown in the preceding table.

Loop
> Select this check box if you want to play the file more than once.

Number of times to play file
> Enter the number of times you want to play the file.

Samples restart
> Select `immediately after last sample` to repeat the audio file immediately. Select `at beginning of next frame` to place the first sample of the audio file in the first position of the next output frame.

Output start-of-file indicator
> Use this check box to determine whether the output contains the first audio sample in the file.

Output end-of-file indicator
> Use this check box to determine whether the output contains the last audio sample in the file.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

See Also

From Audio Device	DSP System Toolbox
Signal From Workspace	DSP System Toolbox
To Multimedia File	DSP System Toolbox
wavread	MATLAB

G711 Codec

Purpose   Quantize narrowband speech input signals

Library   Quantizers

dspquant2
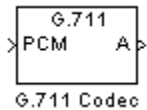
Description   The G711 Codec block is a logarithmic scalar quantizer designed for narrowband speech. Narrowband speech is defined as a voice signal with an analog bandwidth of 4 kHz and a Nyquist sampling frequency of 8 kHz. The block quantizes a narrowband speech input signal so that it can be transmitted using only 8-bits. The G711 Codec block has three modes of operation: encoding, decoding, and conversion. You can choose the block's mode of operation by setting the **Mode** parameter.

If, for the **Mode** parameter, you choose Encode PCM to A-law, the block assumes that the linear PCM input signal has a dynamic range of 13 bits. Because the block always operates in saturation mode, it assigns any input value above $2^{12}-1$ to $2^{12}-1$ and any input value below -2^{12} to -2^{12}. The block implements an A-law quantizer on the input signal and outputs A-law index values. When you choose Encode PCM to mu-law, the block assumes that the linear PCM input signal has a dynamic range of 14 bits. Because the block always operates in saturation mode, it assigns any input value above $2^{13}-1$ to $2^{13}-1$ and any input value below -2^{13} to -2^{13}. The block implements a mu-law quantizer on the input signal and outputs mu-law index values.

If, for the **Mode** parameter, you choose Decode A-law to PCM, the block decodes the input A-law index values into quantized output values using an A-law lookup table. When you choose Decode mu-law to PCM, the block decodes the input mu-law index values into quantized output values using a mu-law lookup table.

If, for the **Mode** parameter, you choose Convert A-law to mu-law, the block converts the input A-law index values to mu-law index values. When you choose Convert mu-law to A-law, the block converts the input mu-law index values to A-law index values.

Note Set the **Mode** parameter to `Convert A-law to mu-law` or `Convert mu-law to A-law` only when the input to the block is A-law or mu-law index values.

If, for the **Mode** parameter, you choose `Encode PCM to A-law` or `Encode PCM to mu-law`, the **Overflow diagnostic** parameter appears on the block parameters dialog box. Use this parameter to determine the behavior of the block when overflow occurs. The following options are available:

- `Ignore` — Proceed with the computation and do not issue a warning message.

- `Warning` — Display a warning message in the MATLAB Command Window, and continue the simulation.

- `Error` — Display an error dialog box and terminate the simulation.

Note Like all diagnostic parameters on the Configuration Parameters dialog box, **Overflow diagnostic** parameter is set to `Ignore` in the code generated for this block by Simulink Coder code generation software.

G711 Codec

Dialog Box
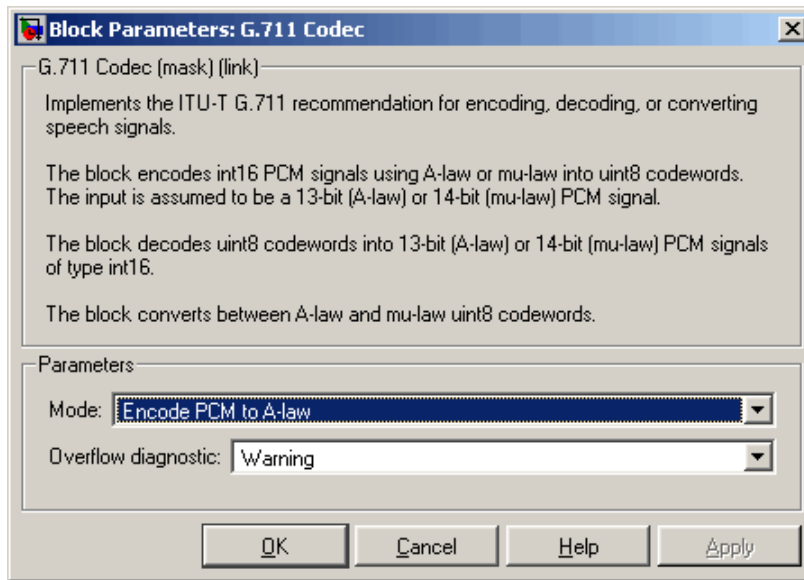
Mode

- When you choose `Encode PCM to A-law`, the block implements an A-law encoder.

- When you choose `Encode PCM to mu-law`, the block implements a mu-law encoder.

- When you choose `Decode A-law to PCM`, the block decodes the input index values into quantized output values using an A-law lookup table.

- When you choose `Decode mu-law to PCM`, the block decodes the input index values into quantized output values using a mu-law lookup table.

- When you choose `Convert A-law to mu-law`, the block converts the input A-law index values to mu-law index values.

- When you choose `Convert mu-law to A-law`, the block converts the input mu-law index values to A-law index values.

Overflow diagnostic

Use this parameter to determine the behavior of the block when overflow occurs.

- Select Ignore to proceed with the computation without a warning message.

- Select Warning to display a warning message in the MATLAB Command Window and continue the simulation.

- Select Error to display an error dialog box and terminate the simulation.

This parameter is only visible if, for the **Mode** parameter, you select Encode PCM to A-law or Encode PCM to mu-law.

References   ITU-T Recommendation G.711, "Pulse Code Modulation (PCM) of Voice Frequencies," *General Aspects of Digital Transmission Systems*; *Terminal Equipments*, International Telecommunication Union (ITU), 1993.

Supported Data Types

Port	Supported Data Types
PCM	• 16-bit signed integers
A	• 8-bit unsigned integers
mu	• 8-bit unsigned integers

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Design	DSP System Toolbox
Uniform Decoder	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Vector Quantizer Decoder	DSP System Toolbox

| Vector Quantizer Design | DSP System Toolbox |
| Vector Quantizer Encoder | DSP System Toolbox |

Purpose   Design halfband filter

Library   Filtering / Filter Designs

                     dspfdesign

Description   This block brings the filter design capabilities of the `filterbuilder`
                     function to the Simulink environment.

```
      Halfband
```

Dialog   See "Halfband Filter Design Dialog Box — Main Pane" on page 7-715
Box   for more information about the parameters of this block. The **Data
                    Types** and **Code Generation** panes are not available for blocks in the
                    DSP System Toolbox Filter Designs library.

Halfband Filter

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter type and order.

Impulse response

Select either FIR or IIR from the drop-down list. FIR is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select either Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Response type

Specify the filter response as Lowpass (the default) or Highpass.

Filter type

Select Single-rate, Decimator, or Interpolator. By default, the block specifies a single-rate filter.

Order

> Enter the filter order. This option is enabled only when the **Filter order mode** is set to `Specify`.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications for a halfband lowpass filter look similar to those shown in the following figure.
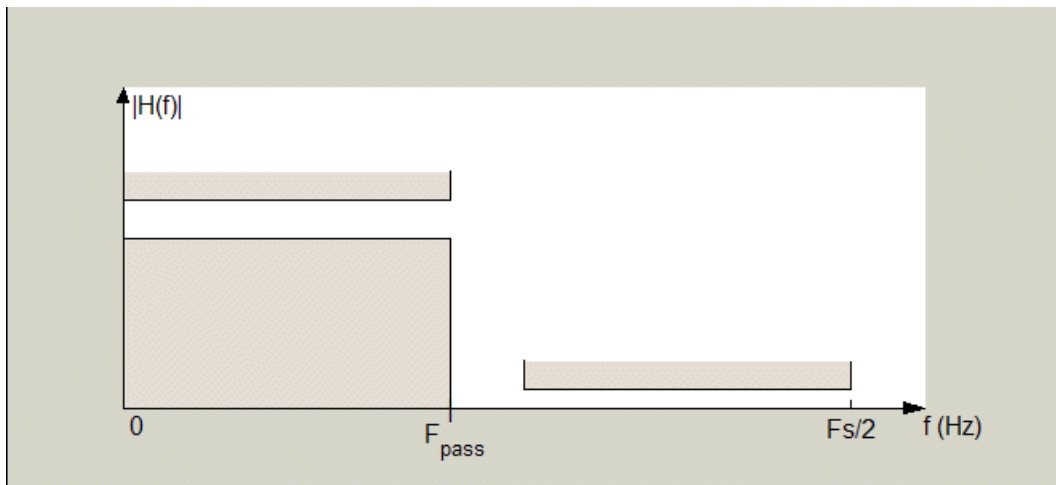
In the figure, the transition region lies between the end of the passband and the start of the stopband. The width is defined explicitly by the value of **Transition width**.

Frequency constraints

> When **Order mode** is `Specify`, set this parameter to `Unconstrained` or `Transition width`.

Frequency units

> Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the

frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude constraints

Specify Unconstrained (the default), or select Stopband attenuation to constrain the response in the stopband explicitly.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

Astop

When **Magnitude units** is Stopband attenuation, enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Halfband Filter

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. For FIR halfband filters, the available design options are `equiripple`, and `Kaiser window`. For IIR halfband filters, the available design options are `Butterworth`, `elliptic`, and `IIR quasi-linear phase`.

Design Options

The following design options are available for FIR halfband filters when the user specifies an equiripple design:

Minimum phase

Select the checkbox to specify a minimum-phase design.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following

conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. The block applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed   see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.

- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

Highpass Filter

Purpose   Design highpass filter

Library   Filtering / Filter Designs

dspfdesign

Description

This block brings the filter design capabilities of the `filterbuilder` function to the Simulink environment.

Dialog Box

See "Highpass Filter Design Dialog Box — Main Pane" on page 7-721 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Highpass Filter

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list. `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select `Minimum` (the default) or `Specify`. Selecting `Specify` enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to `IIR`, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

Order

Enter the filter order. This option is enabled only if you set the **Order mode** to Specify.

Denominator order

Select this check box to specify a different denominator order. This option is enabled only if you set the **Impulse response** to IIR and the **Order mode** to Specify.

Filter type

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

Highpass Filter

In the figure, the region between specification values Fstop and Fpass represents the transition region where the filter response is not constrained.

Frequency constraints

When **Order mode** is `Specify`, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Stopband edge and passband edge` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.

- `Passband edge` — Define the filter by specifying the edge of the passband.

- `Stopband edge` — Define the filter by specifying the edge of the stopband.

- `Stopband edge and 3 dB point` — For IIR filters, define the filter by specifying the frequency of the 3 dB point in the filter response and the edge of the stopband.

- `3 dB point` — Define the filter response by specifying the location of the 3 dB point. The 3 dB point is the frequency for the point three decibels below the passband value.

- `3 dB point and passband edge` — For IIR filters, define the filter by specifying the frequency of the 3 dB point in the filter response and the edge of the passband.

- `6 dB point` — For FIR filters, define the filter response by specifying the location of the 6 dB point. The 6 dB point is the frequency for the point six decibels below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fstop

Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

F3dB

When **Frequency constraints** is `3 dB point`, `Stopband edge and 3 dB point`, or `3 dB point and passband edge`, specify the frequency of the 3 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

F6dB

> When **Frequency constraints** is 6 dB point, specify the frequency of the 6 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude constraints

> This option is only available when you specify the order of your filter design. Depending on the setting of the **Frequency constraints** parameter, some combination of the following options will be available for the **Magnitude constraints** parameter: Unconstrained, Passband ripple, Passband ripple and stopband attenuation or Stopband attenuation.

Magnitude units

> Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Astop

> Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

> Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is `Equiripple`.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

Highpass Filter

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed  see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.

- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option,

you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

Hilbert Filter

Purpose   Design Hilbert filter

Library   Filtering / Filter Designs

                 dspfdesign

Description This block brings the filter design capabilities of the `filterbuilder`
                 function to the Simulink environment.

**Dialog
Box**   See "Hilbert Filter Design Dialog Box — Main Pane" on page 7-729 for
                 more information about the parameters of this block. The **Data Types**
                 and **Code** panes are not available for blocks in the DSP System Toolbox
                 Filter Designs library.

Hilbert Filter

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response

Select either `FIR` or `IIR` from the drop-down list. `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

> **Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the

design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is

available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transitions at the ends of the passband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default)

- Squared — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default FIR method is Equiripple.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

FIR Type

Specify whether to design a type 3 or a type 4 FIR filter. The filter type is defined as follows:

- Type 3 — FIR filter with even order antisymmetric coefficients

- Type 4 — FIR filter with odd order antisymmetric coefficients

Select either 3 or 4 from the drop-down list.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your

filter. By default, FIR filters use `Direct-form FIR`, and IIR filters use `Cascade minimum-multiplier allpass`.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

> **Note** The Inherited (this choice will be removed see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- Enforce single-rate processing — When you select this option, the block maintains the sample rate of the input.

- Allow multirate processing — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to Elements as channels (sample based).

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

Histogram

Purpose   Generate histogram of input or sequence of inputs

Library   Statistics

dspstat3
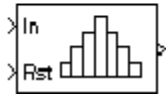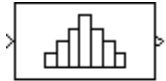
Description   The Histogram block computes the frequency distribution of the
elements in the input. You must use the **Find the histogram over**
parameter to specify whether the block computes the histogram for
Each column of the input or of the Entire input. The **Running
histogram** check box allows you to select between basic operation and
running operation, as described below.

The block distributes the elements of the input into the number of
discrete bins specified by the **Number of bins** parameter, n.

```
y = hist(u,n)     % Equivalent MATLAB code
```

The Histogram block sorts all complex input values into bins according
to their magnitude.

The histogram value for a given bin represents the frequency of
occurrence of the input values bracketed by that bin. You specify
the upper boundary of the highest-valued bin in the **Upper limit of
histogram** parameter, B_M, and the lower boundary of the lowest-valued
bin in the **Lower limit of histogram** parameter, B_m. The bins have
equal width of

$$\Delta = \frac{B_M - B_m}{n}$$

and centers located at

$$B_m + \left(k + \frac{1}{2}\right)\Delta \qquad k = 0,1,2,...,n-1$$

Input values that fall on the border between two bins are placed 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 **Upper limit of histogram** parameter or less than **Lower limit of histogram** parameter are placed into the highest valued or lowest valued bin, respectively.

The values you enter for the **Upper limit of histogram** and **Lower limit of histogram** parameters must be real-valued scalars. `NaN` and `inf` are not valid values for the **Upper limit of histogram** and **Lower limit of histogram** parameters.

Basic Operation

When the **Running histogram** check box is not selected, the Histogram block computes the frequency distribution of the current input.

When you set the **Find the histogram over** parameter to `Each column`, the Histogram block computes a histogram for each column of the *M*-by-*N* matrix independently. The block outputs an *n*-by-*N* matrix, where *n* is the **Number of bins** you specify. The *j*th column of the output matrix contains the histogram for the data in the *j*th column of the *M*-by-*N* input matrix.

When you set the **Find the histogram over** parameter to `Entire input`, the Histogram block computes the frequency distribution for the entire input vector, matrix or N-D array. The block outputs an *n*-by-1 vector, where *n* is the **Number of bins** you specify.

Running Operation

When you select the **Running histogram** check box, the Histogram block computes the frequency distribution of both the past and present data for successive inputs. The block resets the histogram (by emptying all of the bins) when it detects a reset event at the optional Rst port. See "Resetting the Running Histogram" on page 2-754 for more information on how to trigger a reset.

When you set the **Find the histogram over** parameter to `Each column`, the Histogram block computes a running histogram for each column of the *M*-by-*N* matrix. The block outputs an *n*-by-*N* matrix, where *n* is the **Number of bins** you specify. The *j*th column of the

output matrix contains the running histogram for the *j*th column of the *M*-by-*N* input matrix.

When you set the **Find the histogram over** parameter to `Entire input`, the Histogram block computes a running histogram for the data in the first dimension of the input. The block outputs an *n*-by-1 vector, where *n* is the **Number of bins** you specify.

Note When the Histogram block is used in running mode and the input data type is non-floating point, the output of the histogram is stored as a `uint32` data type. The largest number that can be represented by this data type is 2^{32}-1. If the range of the `uint32` data type is exceeded, the output data will wrap back to 0.

Resetting the Running Histogram

The block resets the running histogram whenever a reset event is detected at the optional Rst port. The reset signal and the input data signal must be the same rate.

You specify the reset event using the **Reset port** menu:

- `None` — Disables the Rst port

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

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

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

Rising edge

Rising edge

Rising edge

Rising edge

Not a rising edge because it continues a rise from a negative value to zero.

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

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

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

Falling edge   Falling edge

Falling edge

Falling edge

Not a falling edge because it continues a fall from a positive value to zero.

- `Either edge` — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described earlier)

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

Histogram

Examples　　**Real Input Data**

The bin boundaries created by the Histogram block are determined by the data type of the input:

- Bin boundaries for real, double-precision input are cast into the data type `double`.

- Bin boundaries for real, fixed-point input are cast into the `int8` data type.

The following example shows the differences in the output of the Histogram block based on the data type of the input.

To create this model you need the following blocks.

Block	Library	Quantity
Constant	Simulink /Sources library	2
Display	Sinks	2
Histogram	Statistics	2

The parameter settings for the Double Precision Input Constant block are:

- **Constant value** = `double([1 2 3 4 5]')`

- **Interpret vector parameters as 1-D** = Clear this check box.

- **Sampling mode** = `Sample based`

- **Sample time** = `inf`

The parameter settings for the Fixed-Point Input Constant block are:

- **Constant value** = `int8([1 2 3 4 5]')`

- **Interpret vector parameters as 1-D** = Clear this check box.

- **Sampling mode** = `Sample based`

- **Sample time** = `inf`

The parameter settings for both Histogram blocks are:

- **Lower limit of histogram** = `1`
- **Upper limit of histogram** = `3`
- **Number of bins** = `5`
- **Find the histogram over** = `Entire input`
- **Normalized** = Clear this check box.
- **Running histogram** = Clear this check box.

Connect the blocks as shown in the following figure, and run your model.
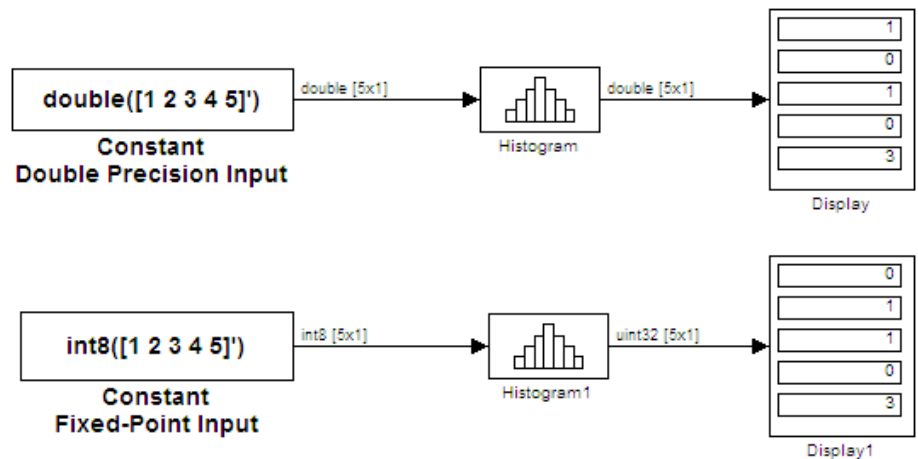
Running the fixed-point model generates the following warning:

```
Warning:  The bin width resulting from the specified
parameters is less than the precision of the input data
type.  This might cause unexpected results.  Since bin
width is calculated by ((upper limit - lower limit)/number
```

of bins), you could increase upper limit or decrease lower
limit or number of bins.

This warning alerts you that it is not a good use case to have a
histogram where 2 or more bin boundaries are the same. As the
warning suggests, increasing the range of the limits of the histogram,
or decreasing the number of bins, can correct this problem.

Dialog Box

The **Main** pane of the Histogram block dialog appears as follows.
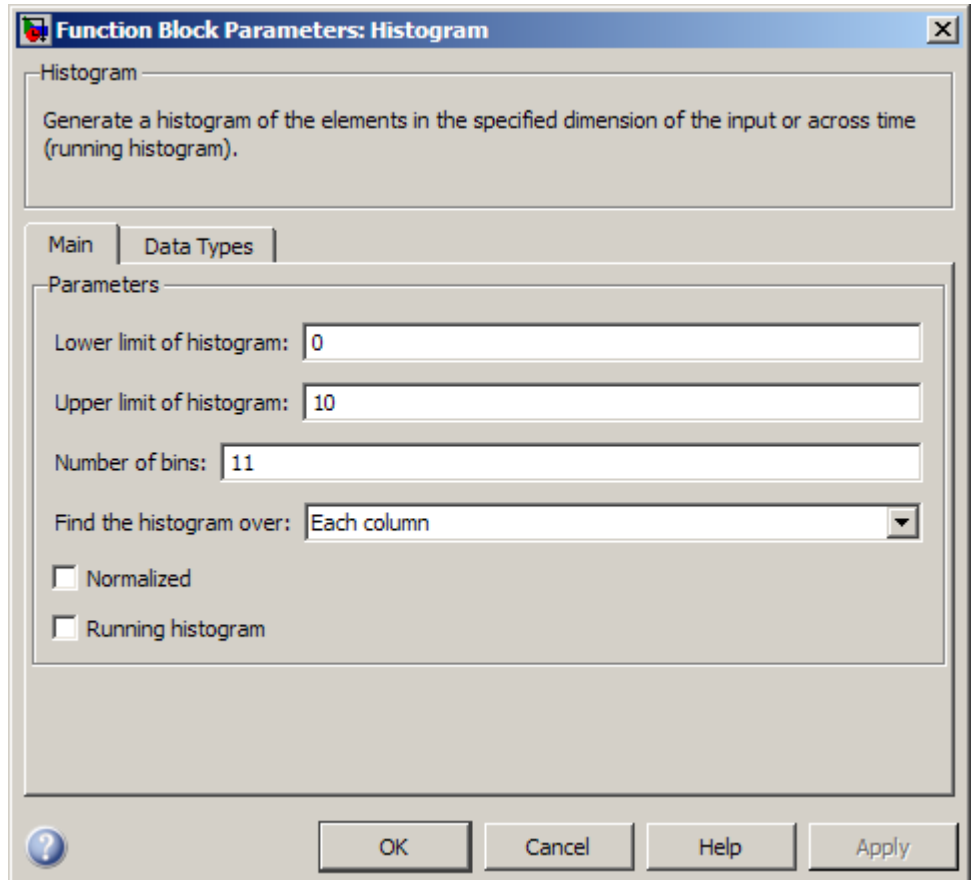
Lower limit of histogram

Enter a real-valued scalar for the lower boundary, B_m, of the lowest-valued bin. NaN and inf are not valid values for B_m. Tunable.

Histogram

Upper limit of histogram

Enter a real-valued scalar for the upper boundary, B_M, of the highest-valued bin. NaN and inf are not valid values for B_M. Tunable.

Number of bins

The number of bins, n, in the histogram.

Find the histogram over

Specify whether the block finds the histogram over the entire input or along each column of the input.

Note The Inherited (this choice will be removed - see release notes) option will be removed in a future release.

Normalized

When selected, the output vector, v, is normalized such that sum(v) = 1.

Use of this parameter is not supported for fixed-point signals.

Running histogram

Set to enable the running histogram operation, and clear to enable basic histogram operation. For more information, see "Basic Operation" on page 2-753 and "Running Operation" on page 2-753.

Reset port

The type of event that resets the running histogram. For more information, see "Resetting the Running Histogram" on page 2-754. The reset signal and the input data signal must be the same rate. This parameter is enabled only when you select the **Running histogram** check box. For more information, see "Running Operation" on page 2-753.

The **Data Types** pane of the Histogram block dialog appears as follows.
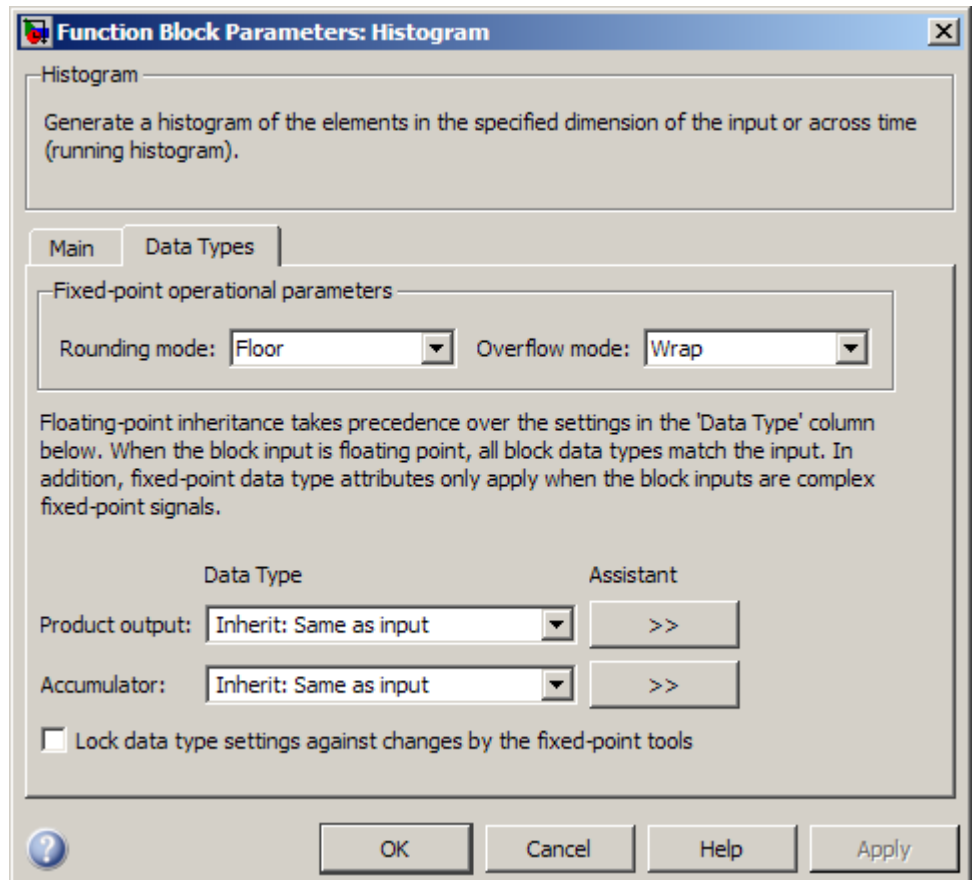
Note The fixed-point parameters listed are only used for fixed-point complex inputs, which are distributed by squared magnitude.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Multiplication Data Types" for illustrations depicting the use of the product output data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button   `>>`   to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button   `>>`   to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• 32-bit unsigned integers
Rst	• Double-precision floating point
	• Single-precision floating point
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Sort	DSP System Toolbox
hist	MATLAB

IDCT

Purpose	Inverse discrete cosine transform (IDCT) of input
Library	Transforms
	dspxfrm3

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
```

When the input is a sample-based row vector, the IDCT block computes the inverse discrete cosine transform across the vector dimension of the input. For all other N-D input arrays, the block computes the IDCT across the first dimension. The size of the first dimension (frame size), must be a power of two. To work with other frame sizes, use the Pad block to pad or truncate the frame size to a power-of-two length.

When the input is an *M*-by-*N* matrix, the block treats each input column as an independent channel containing *M* consecutive samples. The block outputs an *M*-by-*N* matrix whose *l*th column contains the length-*M* IDCT of the corresponding input column.

$$y(m,l) = \sum_{k=1}^{M} w(k)u(k,l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \qquad m = 1,...,M$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{M}}, & k = 1 \\[3mm] \sqrt{\dfrac{2}{M}}, & 2 \leq k \leq M \end{cases}$$

The **Output sampling mode** parameter allows you to select the sampling mode of the output. If the input is an N-D array with 3 or

more dimensions, the **Output sampling mode** must be `Sample based`. The output sample rate and data type (real/complex) are the same as those of the input.

The **Sine and cosine computation** parameter determines how the block computes the necessary sine and cosine values. This parameter has two settings, each with its advantages and disadvantages, as described in the following table.

Sine and Cosine Computation Parameter Setting	Sine and Cosine Computation Method	Effect on Block Performance
`Table lookup`	The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block in a speed-optimized table, and retrieves the values during code execution.	The block usually runs much more quickly, but requires extra memory for storing the precomputed trigonometric values.
`Trigonometric fcn`	The block computes sine and cosine values during the simulation. When you generate code from the block, the processor running the generated code computes the sine and cosine values while the code runs.	The block usually runs more slowly, but does not need extra data memory. For code generation, the block requires a support library to emulate the trigonometric functions, increasing the size of the generated code.

Fixed-Point Data Types

The following diagrams show the data types used within the IDCT block for fixed-point signals. You can set the sine table, accumulator, product

output, and output data types displayed in the diagrams in the IDCT block dialog as discussed in "Dialog Box" on page 2-769.

Inputs to the IDCT block are first cast to the output data type and stored in the output buffer. Each butterfly stage processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type.

Butterfly Stage Data Types

Twiddle Multiplication Data Types

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the

inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types".

Note When the block input is fixed point, all internal data types are signed fixed point.

Dialog Box

The **Main** pane of the IDCT block dialog appears as follows.
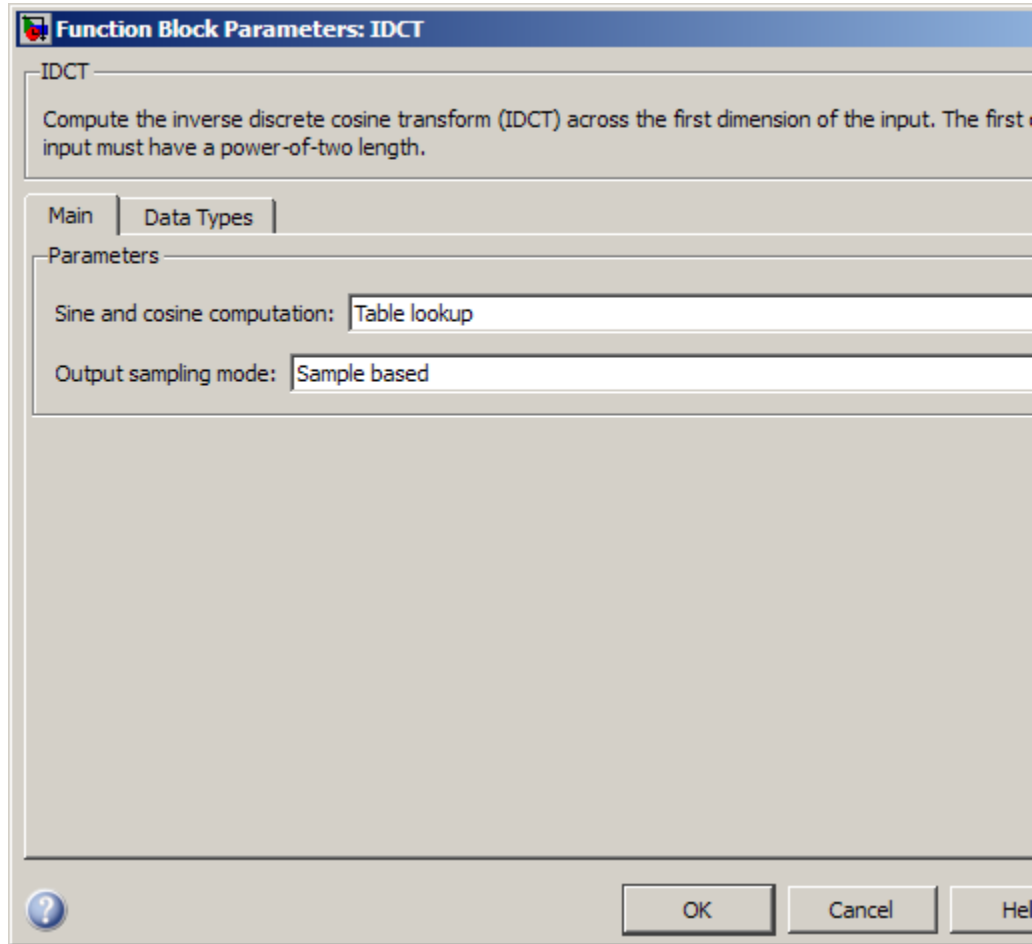
Sine and cosine computation

Sets the block to compute sines and cosines by either looking up sine and cosine values in a speed-optimized table (Table lookup),

or by making sine and cosine function calls (`Trigonometric fcn`).

See the table in the "Description " on page 2-764 section.

Output sampling mode

Select `Sample based` or `Frame based` output. If the input to the IDCT block has 3 or more dimensions, you must select `Sample based` output.

The **Data Types** pane of the IDCT block dialog appears as follows.

Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; instead, they are always saturated and rounded to `Nearest`.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-765 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-765 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-765 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.

  When you select `Inherit:  Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The internal rule first calculates an ideal output word length and fraction length using the following equations:

  $$WL_{ideal\ output} = WL_{input} + floor(\log_2(DCT\ length - 1)) + 1$$

  $$FL_{ideal\ output} = FL_{input}$$

  Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your

hardware. For more information, see "Inherit via Internal Rule".

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

See Also

DCT	DSP System Toolbox
IFFT	DSP System Toolbox
idct	Signal Processing Toolbox

Identity Matrix

Purpose   Generate matrix with ones on main diagonal and zeros elsewhere

Library
- Sources

  dspsrcs4

- Math Functions / Matrices and Linear Algebra / Matrix Operations

  dspmtrx3

Description

The Identity Matrix block generates a rectangular matrix with ones on the main diagonal and zeros elsewhere.

When you select the **Inherit output port attributes from input port** check box, the input port is enabled, and an M-by-N matrix input generates an M-by-N matrix output with the same sample period as the input. The values in the input matrix are ignored.

```
y = eye([M N])      % Equivalent MATLAB code
```

When you do not select the **Inherit output port attributes from input port** check box, the input port is disabled, and the dimensions of the output matrix are determined by the **Matrix size** parameter. A scalar value, M, specifies an M-by-M identity matrix, while a two-element vector, [M N], specifies an M-by-N unit-diagonal matrix. You can specify the output sample period using the **Sample time** parameter.

Examples   Set **Matrix size** to [3 6] to generate the 3-by-6 unit-diagonal matrix below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Dialog Box

The **Main** pane of the Identity Matrix block dialog appears as follows.
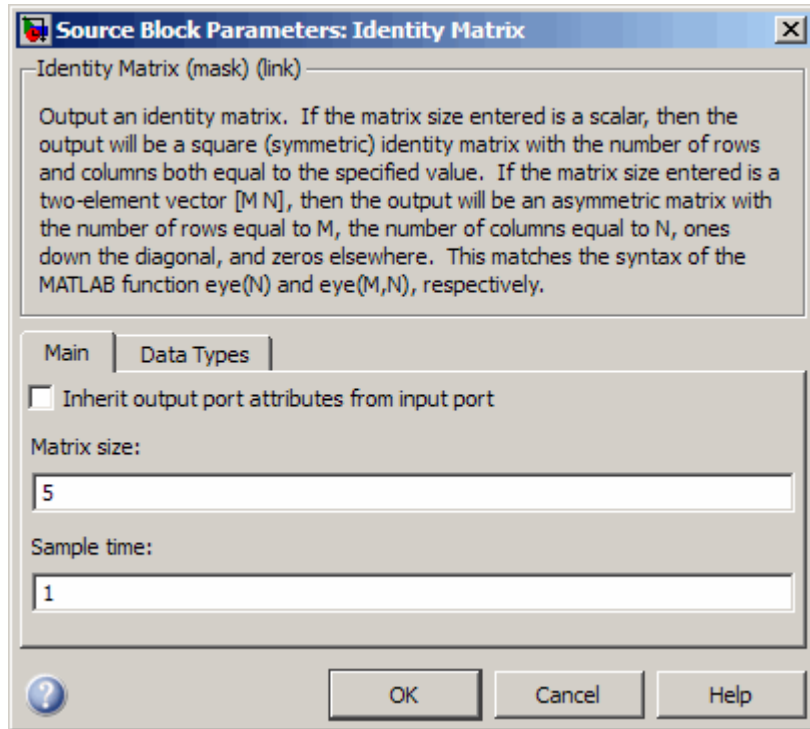
Inherit output port attributes from input port

Enables the input port when selected. In this mode, the output inherits its dimensions, sample period, and data type from the input. The output is always real.

Matrix size

The number of rows and columns in the output matrix: a scalar *M* for a square *M*-by-*M* output, or a vector [*M N*] for an *M*-by-*N* output. This parameter is disabled when you select **Inherit input port attributes from input port**.

Identity Matrix

Sample time

> The discrete sample period of the output. This parameter is disabled when you select **Inherit input port attributes from input port**.

The **Data Types** pane of the Identity Matrix block dialog appears as follows.

Output data type

> Specify the output data type for this block. You can select one of the following:
>
> - A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`. When you select this option,

the output data type and scaling matches that of the next downstream block.

- A built in data type, such as `double`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Identity Matrix

See Also

Constant Diagonal Matrix	DSP System Toolbox
Constant	Simulink
`eye`	MATLAB

Purpose Inverse discrete wavelet transform (IDWT) of input or reconstruct signals from subbands with smaller bandwidths and slower sample rates

Library Transforms

dspxfrm3
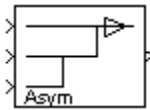
Description The IDWT block is the same as the Dyadic Synthesis Filter Bank block in the Multirate Filters library, but with different default settings. See the Dyadic Synthesis Filter Bank block reference page for more information on how to use the block.
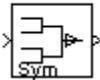
IFFT

Purpose	Inverse fast Fourier transform (IFFT) of input
Library	Transforms
	dspxfrm3

Description   The IFFT block computes the inverse fast Fourier transform (IFFT) of each row of a sample-based 1-by-*P* input vector, or across the first dimension (*P*) of an *N-D* input array.

When you specify an FFT length not equal to the length of the input vector, (or first dimension of the input array), the block implements zero padding or modulo-*M*, (FFT length) data wrapping. This occurs before the IFFT operation, as per Orfanidis [1].

```
y = ifft(u,M)  % P   M


y(:,l) = ifft(datawrap(u(:,l),M))  % P > M; l = 1,...,N
```

When the input length, *P*, is greater than the FFT length, *M*, you may see magnitude increases in your IFFT output. These magnitude increases occur because the IFFT block uses modulo-*M* data wrapping to preserve all available input samples.

To avoid such magnitude increases, you can truncate the length of your input sample, *P*, to the FFT length, *M*. To do so, place a Pad block before the IFFT block in your model.

The *k*th entry of the *l*th output channel, $y(k, l)$, is equal to the *k*th point of the *M*-point inverse discrete Fourier transform (IDFT) of the *l*th input channel:

$$y(k,l) = \frac{1}{M} \sum_{p=1}^{P} u(p,l) e^{j2\pi(p-1)(k-1)/M} \qquad k = 1,\ldots,M$$

The output of this block has the same dimensions as the input. If the input signal has a floating-point data type, the data type of the output signal uses the same floating-point data type. Otherwise, the

2-782

output can be any fixed-point data type. The block computes scaled and unscaled versions of the IFFT.

The input to this block can be floating-point or fixed-point, real or complex, and conjugate symmetric. The block uses one of two possible FFT implementations. You can select an implementation based on the FFTW library [1], [2], or an implementation based on a collection of Radix-2 algorithms. You can select Auto to allow the block to choose the implementation.

FFTW Implementation

The FFTW implementation provides an optimized FFT calculation including support for power-of-two and non-power-of-two transform lengths in both simulation and code generation. Generated code using the FFTW implementation will be restricted to MATLAB host computers. The data type must be floating-point. Refer to Simulink Coder for more details on generating code.

Radix-2 Implementation

The Radix-2 implementation supports bit-reversed processing, fixed or floating-point data, and allows the block to provide portable C-code generation using the Simulink Coder. The dimension M of the M-by-N input matrix, must be a power of two. To work with other input sizes, use the Pad block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation.

With Radix-2 selected, the block implements one or more of the following algorithms:

- Butterfly operation
- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm

Radix-2 Algorithms for Real or Complex Input Complexity

Parameter Settings	Algorithms Used for IFFT Computation
☐ Input is in bit-reversed order ☐ Input is conjugate symmetric	Bit-reversal operation and radix-2 DIT
☑ Input is in bit-reversed order ☐ Input is conjugate symmetric	Radix-2 DIT
☐ Input is in bit-reversed order ☑ Input is conjugate symmetric	Bit-reversal operation and radix-2 DIT in conjunction with the half-length and double-signal algorithms
☑ Input is in bit-reversed order ☑ Input is conjugate symmetric	Radix-2 DIT in conjunction with the half-length and double-signal algorithms

Radix-2 Optimization for the Table of Trigonometric Values

In certain situations, the block's Radix–2 algorithm computes all the possible trigonometric values of the twiddle factor

$$e^{j\frac{2\pi k}{K}}$$

where K is the greater value of either M or N and $k = 0, \cdots, K-1$. The block stores these values in a table and retrieves them during simulation. The number of table entries for fixed-point and floating-point is summarized in the following table:

Number of Table Entries for N-Point FFT	
floating-point	3 $N/4$
fixed-point	N

Fixed-Point Data Types

The following diagrams show the data types used in the IFFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the IFFT dialog box as discussed in "Dialog Box" on page 2-790.

Inputs to the IFFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. The block multiplies in a twiddle factor before each butterfly stage in a decimation-in-time IFFT and after each butterfly stage in a decimation-in-frequency IFFT.

Decimation-in-time IFFT

Decimation-in-frequency IFFT
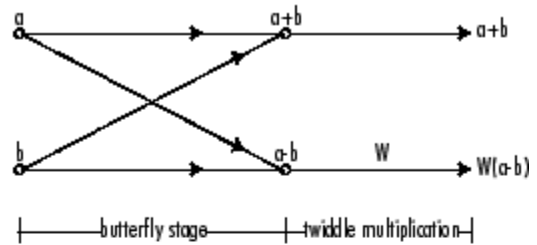
Butterfly stage data types
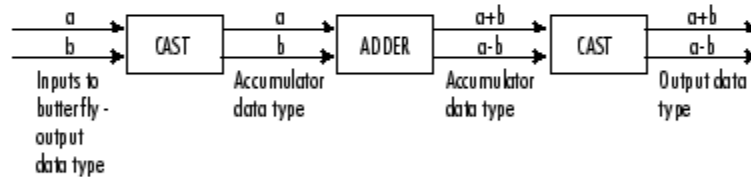
Twiddle multiplication data types

The multiplier output appears in the accumulator data type because both of the inputs to the multiplier are complex. For details on the complex multiplication performed, refer to "Multiplication Data Types" in the DSP System Toolbox documentation.

Fixed-Point Data Types

The following diagrams show the data types used within the IFFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the IFFT block dialog, as discussed in "Dialog Box" on page 2-790.

The IFFT block first casts input to the output data type and then stores it in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. The block multiplies in a twiddle factor before each butterfly stage in a decimation-in-time IFFT, and after each butterfly stage in a decimation-in-frequency IFFT.

Decimation-in-time IFFT
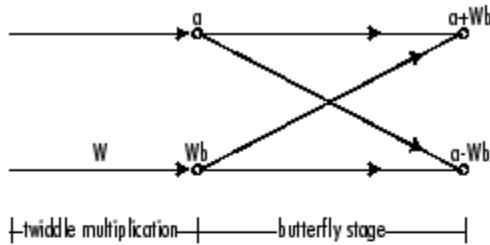
Decimation-in-frequency IFFT

Butterfly stage data types
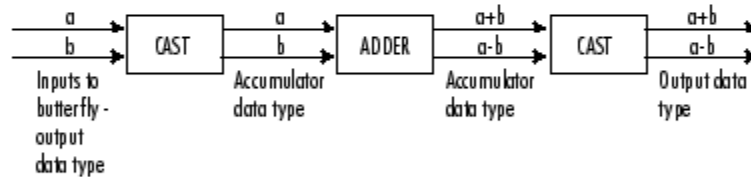
Twiddle multiplication data types

The output of the multiplier is in the accumulator data type because both of the inputs to the multiplier are complex. For details on the complex multiplication performed, see "Multiplication Data Types".

Note When the block input is fixed point, all internal data types are signed fixed point.

IFFT

The **Main** pane of the IFFT block dialog appears as follows.
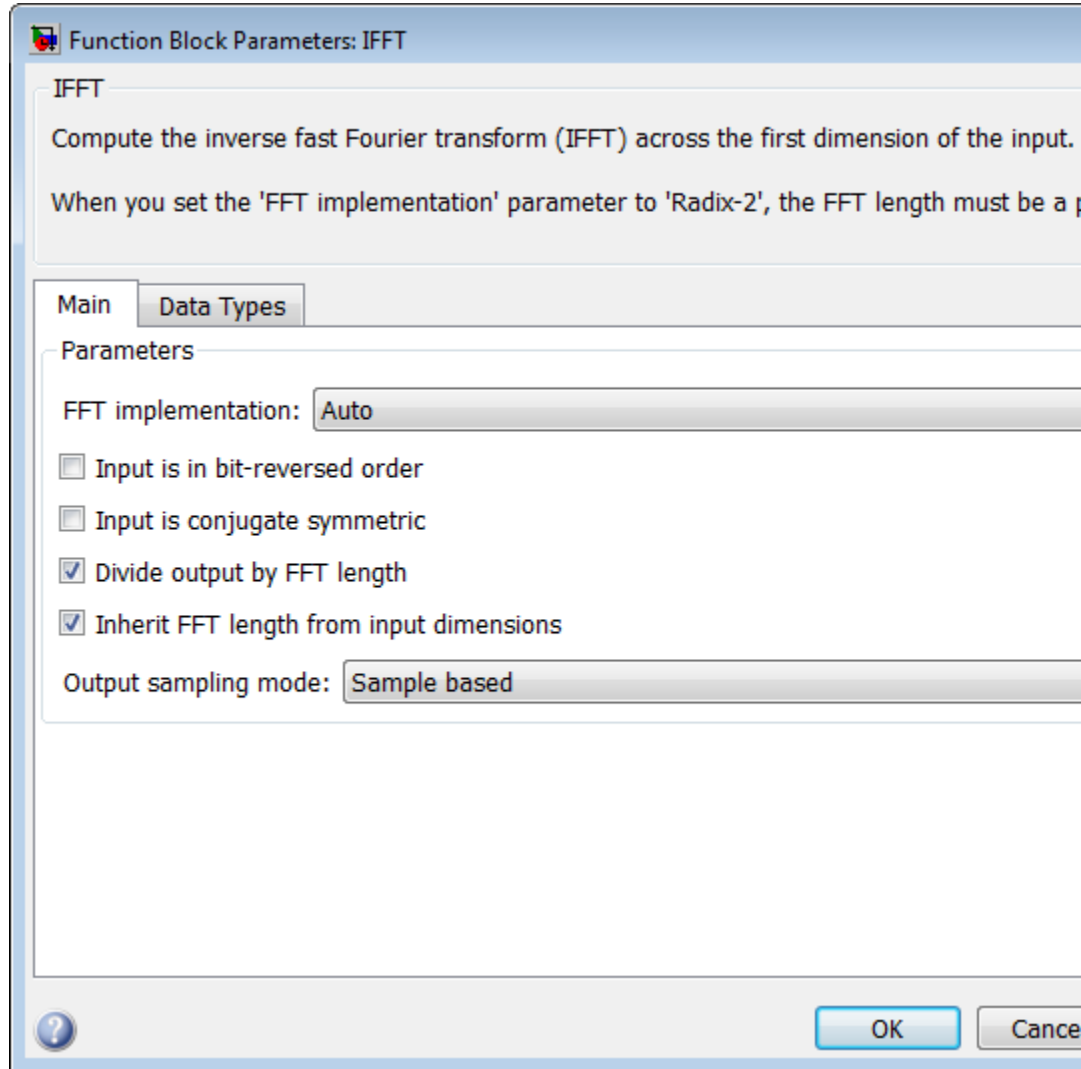
Function Block Parameters: IFFT

IFFT

Compute the inverse fast Fourier transform (IFFT) across the first dimension of the input.

When you set the 'FFT implementation' parameter to 'Radix-2', the FFT length must be a p

Main | Data Types

Parameters

FFT implementation: Auto

☐ Input is in bit-reversed order

☐ Input is conjugate symmetric

☑ Divide output by FFT length

☑ Inherit FFT length from input dimensions

Output sampling mode: Sample based

OK   Cance

FFT implementation

Set this parameter to FFTW [1], [2] to support an arbitrary length input signal. The block restricts generated code with FFTW implementation to MATLAB host computers.

Set this parameter to Radix-2 for bit-reversed processing, fixed or floating-point data, or for portable C-code generation using the Simulink Coder. The dimension *M* of the *M*-by-*N* input matrix, must be a power of two. To work with other input sizes, use the Pad block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation. See "Radix-2 Implementation" on page 2-783.

Set this parameter to Auto to let the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

Input is in bit-reversed order

Select or clear this check box to designate the order of the input channel elements. Select this check box when the input should appear in reversed order, and clear it when the input should appear in linear order. The block yields invalid outputs when you do not set this parameter correctly. This check box only appears when you set the **FFT implementation** parameter to Radix-2 or Auto.

You cannot select this check box if you have cleared the **Inherit FFT length from input dimensions** check box, and you are specifying the FFT length using the **FFT length** parameter. Also, it cannot be selected when you set the **FFT implementation** parameter to FFTW.

For more information on ordering of the output, see "Linear and Bit-Reversed Output Order".

Input is conjugate symmetric

>Select this option when the block inputs conjugate symmetric data and you want real-valued outputs. Selecting this check box optimizes the block's computation method.

>The FFT block yields conjugate symmetric output when you input real-valued data. Taking the IFFT of a conjugate symmetric input matrix produces real-valued output. Therefore, if the input to the block is both floating point and conjugate symmetric, and you select the this check box, the block produces real-valued outputs.

>You cannot select this check box if you have cleared the **Inherit FFT length from input dimensions** check box, and you are specifying the FFT length using the **FFT length** parameter.

>If you input conjugate symmetric data to the IFFT block and do not select this check box, the IFFT block outputs a complex-valued signal with small imaginary parts. The block outputs invalid data if you select this option with non conjugate symmetric input data.

Divide output by FFT length

>When you select this check box, the block computes its output according to the IDFT equation, discussed in the Description section.

>When you clear this check box, the block computes the output using a modified version of the IDFT: $M \cdot y(k, l)$, which is defined by the following equation:

$$M \cdot y(k, l) = \sum_{p=1}^{P} u(p, l) e^{j2\pi(p-1)(k-1)/M} \quad k = 1, ..., M$$

>Notice, the modified IDFT equation does not include the multiplication factor of $1/M$.

Inherit FFT length from input dimensions

Select to inherit the FFT length from the input dimensions. If you do not select this parameter, the **FFT length** parameter becomes available to specify the length. You cannot clear this parameter when you select either the **Input is in bit-reversed order** or the **Input is conjugate symmetric** parameter.

FFT length

Specify FFT length. This parameter only becomes available if you do not select the **Inherit FFT length from input dimensions** parameter.

When you set the **FFT implementation** parameter to `Radix-2`, or when you check the **Output in bit-reversed order** check box, this value must be a power of two.

o

Output sampling mode

Select `Sample based` or `Frame based` output. If the input to the IFFT block has 3 or more dimensions, you must select `Sample based` output.

The **Data Types** pane of the IFFT block dialog appears as follows.

Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; instead, they always round to `Nearest`.

Overflow mode

Select the overflow mode for fixed-point operations. The sine table values do not obey this parameter; instead, they are always saturated.

Sine table data type

Choose how you specify the word length of the values of the sine table. The fraction length of the sine table values always equals the word length minus one. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same word length as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; instead, they are always saturated and rounded to `Nearest`.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-787 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button $\boxed{\quad >> \quad}$ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-787 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-787 for illustrations depicting the use of the output data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`.

  When you select `Inherit:  Inherit via internal rule`, the block calculates the output word length and fraction length automatically. The equations that the block uses to calculate the ideal output word length and fraction length depend on the setting of the **Divide output by FFT length** check box.

  – When you select the **Divide output by FFT length** check box, the ideal output word and fraction lengths are the same as the input word and fraction lengths.

— When you clear the **Divide output by FFT length** check box, the block computes the ideal output word and fraction lengths according to the following equations:

$$WL_{ideal\ output} = WL_{input} + floor(\log_2(FFT\ length - 1)) + 1$$

$$FL_{ideal\ output} = FL_{input}$$

Using these ideal results, the internal rule then selects word lengths and fraction lengths that are appropriate for your hardware. For more information, see "Inherit via Internal Rule".

• An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Examples   See "Transform Frequency-Domain Data into Time Domain" in the *DSP System Toolbox User's Guide*.

References   [1] Orfanidis, S. J. *Introduction to Signal Processing.* Upper Saddle River, NJ: Prentice Hall, 1996, p. 497.

[2] Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996.

IFFT

[3] FFTW (http://www.fftw.org)

[4] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

References

[1] FFTW (http://www.fftw.org)

[2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

See Also

FFT	DSP System Toolbox
IDCT	DSP System Toolbox
Pad	DSP System Toolbox
bitrevorder	Signal Processing Toolbox

fft	MATLAB
ifft	MATLAB

Inherit Complexity

Purpose   Change complexity of input to match reference signal

Library   Signal Management / Signal Attributes

dspsigattribs
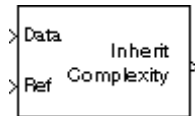
Description   The Inherit Complexity block alters the input data at the Data port to match the complexity of the reference input at the Ref port. When the Data input is real, and the Ref input is complex, the block appends a zero-valued imaginary component, 0i, to each element of the Data input.
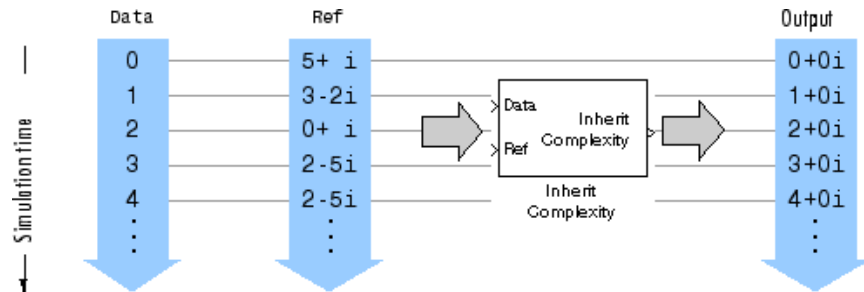
When the Data input is complex, and the Ref input is real, the block outputs the real component of the Data input.
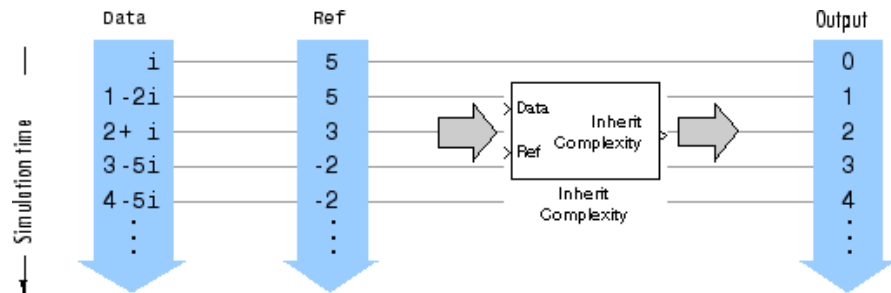
When both the Data input and Ref input are real, or when both the Data input and Ref input are complex, the block propagates the Data input with no change.

Dialog Box

Inherit Complexity (mask) (link)

Copy data from the Data input with the complexity of the reference signal. If the data is real and the reference is complex, an all zero imaginary part is created. If the data is complex and the reference is real, the imaginary part of the input is removed.

OK   Cancel   Help   Apply
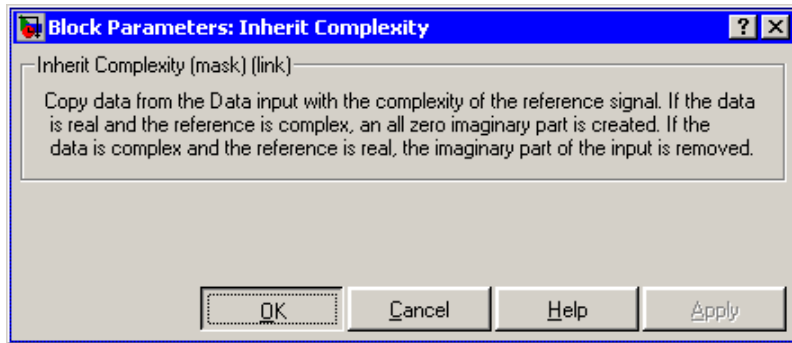
Supported Data Types

Port	Supported Data Types
Data	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Ref	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Inherit Complexity

Port	Supported Data Types
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Check Signal Attributes	DSP System Toolbox
Complex to Magnitude-Angle	Simulink
Complex to Real-Imag	Simulink
Magnitude-Angle to Complex	Simulink
Real-Imag to Complex	Simulink

Purpose   Delay input by integer number of sample periods

Library   dspobslib

Description
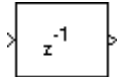
> **Note** The Integer Delay block will be removed from the product in a future release. We strongly recommend replacing this block with the Delay block.

The Integer Delay block delays a discrete-time input by the number of sample intervals specified in the **Delay** parameter. Noninteger delay values are rounded to the nearest integer, and negative delays are clipped at 0.

Sample-Based Operation

When the input is a sample-based M-by-N matrix, the block treats each of the $M*N$ matrix elements as an independent channel. The **Delay** parameter, v, can be an M-by-N matrix of positive integers that specifies the number of sample intervals to delay each channel of the input, or a scalar integer by which to equally delay all channels.

For example, when the input is M-by-1 and v is the matrix [v(1) v(2) ... v(M)]', the first channel is delayed by v(1) sample intervals, the second channel is delayed by v(2) sample intervals, and so on. Note that when a channel is delayed for Δ sample-time units, the output sample at time t is the input sample at time $t - \Delta$. When $t - \Delta$ is negative, then the output is the corresponding value specified by the **Initial conditions** parameter.

A 1-D vector of length M is treated as an M-by-1 matrix, and the output is 1-D.

The **Initial conditions** parameter specifies the output of the block during the initial delay in each channel. The initial delay for a particular channel is the time elapsed from the start of the simulation until the first input in that channel is propagated to the output. Both

Integer Delay (Obsolete)

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,jth element of the kth matrix in the input sequence.

- Array of size M-by-N-by-d. In this case, you can set different fixed initial conditions for each element of a sample-based input. This setting is explained further in the Array bullet in "Time-Varying Initial Conditions" on page 2-804.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions

A time-varying initial condition in sample-based mode can be specified in one of the following ways:

- Vector of length d, where d is the maximum value specified for any channel in the **Delay** parameter. The vector can be a L-by-d, 1-by-d, or 1-by-1-by-d. The d elements of the vector are output in sequence, one at each sample time of the initial delay.

  For a scalar input and the parameters shown below,

  Delay (samples):

  5

  Initial conditions:

  [-1 -1 -1 0 1]

  the block outputs the sequence -1, -1, -1, 0, 1,... at the start of the simulation.

- Array of dimension M-by-N-by-d, where d is the value specified for the **Delay** parameter (the maximum value when the **Delay** is a vector) and M and N are the number of rows and columns, respectively, in the input matrix. The d pages of the array are output in sequence, one at each sample time of the initial delay. For a 2-by-3 input, and the parameters below,

  Delay (samples):

  3

  Initial conditions:

  cat(3, [1 2 3; 4 5 6], [2 4 6; 1 3 5], [3 6 9; 0 4 8])

  the block outputs the matrix sequence

  $$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \begin{bmatrix} 2 & 4 & 6 \\ 1 & 3 & 5 \end{bmatrix}, \begin{bmatrix} 3 & 6 & 9 \\ 0 & 4 & 8 \end{bmatrix}$$

  at the start of the simulation. Note that setting **Initial conditions** to an array with the same matrix for each entry implements constant initial conditions; a different constant initial condition for each input matrix element (channel).

Initial conditions cannot be specified by full matrices.

Integer Delay (Obsolete)

Frame-Based Operation

When the input is a frame-based M-by-N matrix, the block treats each of the N columns as an independent channel, and delays each channel as specified by the **Delay** parameter.

For frame-based inputs, the **Delay** parameter can be a scalar integer by which to equally delay all channels. It can also be a 1-by-N row vector, each element of which serves as the delay for the corresponding channel of the N-channel input. Likewise, it can also be an M-by-1 column vector, each element of which serves as the delay for one of the corresponding M samples for each channel. The **Delay** parameter can be an M-by-N matrix of positive integers as well; in this case, each element of each channel is delayed by the corresponding element in the delay matrix. For instance, if the fifth element of the third column of the delay matrix was 3, then the fifth element of the third channel of the input matrix is always delayed by three sample-time units.

When a channel is delayed for Δ sample-time units, the output sample at time t is the input sample at time $t - \Delta$. When $t - \Delta$ is negative, then the output is the corresponding value specified in the **Initial conditions** parameter.

The **Initial conditions** parameter specifies the output during the initial delay. Both fixed and time-varying initial conditions can be specified. The initial delay for a particular channel is the time elapsed from the start of the simulation until the first input in that channel is propagated to the output.

Fixed Initial Conditions

The settings shown below specify fixed initial conditions. The value entered in the **Initial conditions** parameter is repeated at the output for each sample time of the initial delay. A fixed initial condition in frame-based mode can be one of the following:

- Scalar value to be repeated for all channels of the output at each sample time of the initial delay. For a general M-by-N input with the parameter settings below,

Delay (samples):

5

Initial conditions:

0

the first five samples in each of the *N* channels are zero. Notice that when the frame size is larger than the delay, all of these zeros are all included in the first output from the block.

- Array of size 1-by-*N*-by-*D*. In this case, you can also specify different fixed initial conditions for each channel. See "Time-Varying Initial Conditions" on page 2-807 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 are 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,

Integer Delay (Obsolete)

the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -4 & -1 \\ -5 & -2 \\ 1 & -3 \\ 2 & -4 \end{bmatrix}, \begin{bmatrix} 3 & -5 \\ 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{bmatrix}, \begin{bmatrix} 7 & 4 \\ 8 & 5 \\ 9 & 6 \\ 10 & 7 \end{bmatrix}, \dots$$

Note that since one of the delays, 2, is less than the frame size of the input, 4, the length of the **Initial conditions** vector is the sum of the maximum delay and 1 (5+1), which is 6. The first five entries of the initial conditions vector are used by the channel with the maximum delay, and the rest of the entries are ignored. Since the first channel is delayed for less than the maximum delay (2 sample time units), it only makes use of two of the initial condition entries.

- Array of size 1-by-N-by-D, where D is defined in "Time-Varying Initial Conditions" on page 2-807. In this case, the kth entry of each 1-by-N entry in the array corresponds to an initial condition for the kth channel of the input matrix. Thus, a 1-by-N-by-D initial conditions input allows you to specify different initial conditions for each channel. For instance, for a two-channel ramp input [1:100; 1:100]' with a frame size of 4 and the parameter settings below,
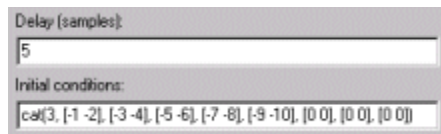
the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -2 \\ -3 & -4 \\ -5 & -6 \\ -7 & -8 \end{bmatrix}, \begin{bmatrix} -9 & -10 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

Note that the channels have distinct time varying initial conditions; the initial conditions for channel 1 correspond to the first entry of each length-2 row vector in the initial conditions array, and the initial conditions for channel 2 correspond to the second entry of each row vector in the initial conditions array. Only the first five entries in the initial conditions array are used; the rest are ignored.

The 1-by-N-by-D array entry can also specify different fixed initial conditions for every channel; in this case, every 1-by-N entry in the array would be identical, so that the initial conditions for each column are fixed over time.

Initial conditions cannot be specified by full matrices.

Resetting the Delay

The block resets the delay whenever it detects a reset event at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

You specify the reset event in the **Reset port** parameter:

- None disables the Rst port.

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

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

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

Rising edge

Rising edge

Rising edge

Rising edge

Not a rising edge because it continues a rise from a negative value to zero.

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

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

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

Falling edge   Falling edge

Falling edge

Falling edge

Not a falling edge because it continues a fall from a positive value to zero.

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

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

Note When running simulations in the Simulink MultiTasking mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

Examples   The dspafxr example illustrates an audio reverberation system built around the Integer Delay block.

Dialog Box
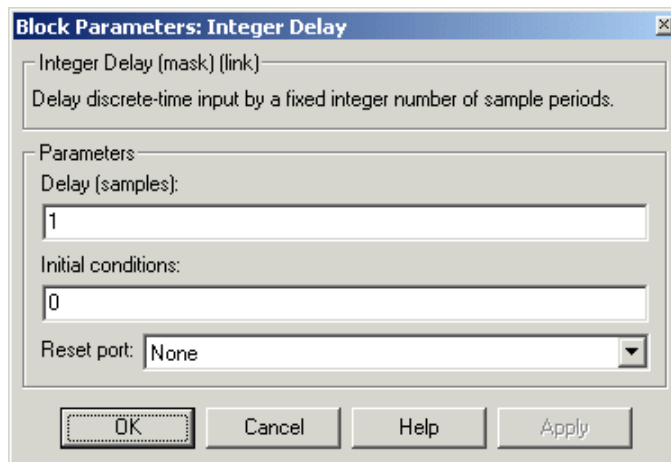
Delay
:   The number of sample periods to delay the input signal.

Initial conditions
:   The value of the block's output during the initial delay.

Integer Delay (Obsolete)

Reset port
> Determines the reset event that causes the block to reset the delay. For more information, see "Resetting the Delay" on page 2-809.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- Boolean — The block accepts Boolean inputs to the Rst port, which is enabled by the **Reset port** parameter.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

Unit Delay	Simulink
Variable Fractional Delay	DSP System Toolbox
Variable Integer Delay	DSP System Toolbox

Interpolation

Purpose   Interpolate values of real input samples

Library   Signal Operations

dspsigops
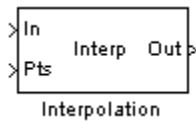
Description   The Interpolation block interpolates discrete, real, inputs using linear
or FIR interpolation. The block accepts both sample- and frame-based
input data in the form of a vector, matrix, or sample-based N-D
array. The block outputs a scalar, vector, matrix, or N-D array of the
interpolated values, which has the same frame status as the input data.

You must specify the interpolation points (times at which to interpolate
values) in a one-based interpolation array, I_{Pts}. An entry of 1 in I_{Pts}
refers to the first sample of the input data, an entry of 2.5 refers to the
sample half-way between the second and third input sample, and so on.
Depending on the dimensions of the input data, I_{Pts} can be a scalar,
a length-P row or column vector, a P-by-N frame-based matrix, or a
sample-based N-D array where P is the size of the first dimension of
the N-D array. In most cases, P can be any positive integer. For more
information about valid interpolation arrays, refer to the tables in "How
the Block Applies Interpolation Arrays to Inputs" on page 2-814.

In most cases, the block applies I_{Pts} across the first dimension of an N-D
input array, or to each input vector. You can set the block to apply the
same interpolation array for all input data (static interpolation points
entered on the block mask) or to use a different interpolation array for
each N-D array, matrix, or vector input (time-varying interpolation
points received via the Pts input port).

Sections of This Reference Page

Interpolation

- "FIR Interpolation Mode" on page 2-822

- "Dialog Box" on page 2-823

- "Supported Data Types" on page 2-826

Specifying Static Interpolation Points

To supply the block with a static interpolation array (an interpolation array applied to every vector or N-D array of input data), perform the following steps:

- Set the **Source of interpolation points** parameter to `Specify via dialog`.

- Enter the interpolation array in the **Interpolation points** parameter. To learn about interpolation arrays, see "How the Block Applies Interpolation Arrays to Inputs" on page 2-814.

Specifying Time-Varying Interpolation Points

To supply the block with time-varying interpolation arrays (where the block uses a different interpolation array for each vector or N-D array input), perform the following steps:

1 Set the **Source of interpolation points** parameter to `Input port`, the **Pts** port appears on the block.

2 Generate a signal of interpolation arrays, and supply it to the Pts port. The block uses the input to this port as the interpolation points. To learn about interpolation arrays, see "How the Block Applies Interpolation Arrays to Inputs" on page 2-814.

How the Block Applies Interpolation Arrays to Inputs

The interpolation array I_{Pts} represents the points in time at which to interpolate values of the input signal. An entry of 1 in I_{Pts} refers to the first sample of the input, an entry of 2.5 refers to the sample half-way between the second and third input sample, and so on. In most cases, when I_{Pts} is a vector, it can be of any length.

Valid values in the interpolation array, I_{Pts}, range from 1 to the number of samples in each channel of the input. To learn how the block handles out of range interpolation values, see "Handling Out-of-Range Interpolation Points" on page 2-820.

Depending on the dimension and frame status of the input and the dimension of I_{Pts}, the block usually applies I_{Pts} to the input in one of the following ways:

- Applies the I_{Pts} array across the first dimension of a sample-based N-D array or frame-based matrix input, resulting in a sample-based N-D array or frame-based matrix output.

- Applies the vector I_{Pts} to each input vector (as if the input vector were a single channel), resulting in a vector output with the same orientation as the input (row or column).

The following tables summarize how the block applies the interpolation array I_{Pts} to all the possible types of sample- and frame-based inputs, and show the resulting output dimensions.

The first table describes the block's behavior when the **Source of interpolation points** is `Specify via dialog` and the input is sample based.

Input Dimensions (Sample Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Sample Based)
N-D Array (ex. M-by-N-by-Q)	1-by-P row	Applies I_{Pts} to the first dimension of the input array	P-by-N-by-Q array
	P-by-1 column		
	P-by-N-by-Q array	Applies the columns of I_{Pts} to the corresponding	

Interpolation

Input Dimensions (Sample Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Sample Based)
		columns of the input array	
M-by-1 column	1-by-P row (block treats I_n as a column)	Applies I_{Pts} to the input column	P-by-1 column
	P-by-1 column		
1-by-N row	1-by-P row	Applies I_n to the input row	1-by-P row
	P-by-1 column (block treats I_{Pts} as a row)		

The next table describes the block's behavior when the **Source of interpolation points** is Specify via dialog and the input is frame based.

Input Dimensions (Frame Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Frame Based)
M-by-N matrix	1-by-N row	Applies each column of I_{Pts} (each element of I_{Pts}) to the corresponding column of the input matrix	1-by-N row

Input Dimensions (Frame Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Frame Based)
	P-by-1 column	Applies I_{Pts} to each input column	P-by-N matrix
	P-by-N matrix	Applies the columns of I_{Pts} to the corresponding columns of the input matrix	
M-by-1 column	1-by-P row (block treats I_{Pts} as a column)	Applies I_{Pts} to the input column	P-by-1 column
	P-by-1 column		
1-by-N row (not recommended)	1-by-N row	Not Applicable. Block copies input vector	1-by-N row, a copy of the input vector
	P-by-1 column		P-by-N matrix where each row is a copy of the input vector
	P-by-N matrix		

The next table describes the block's behavior when the **Source of interpolation points** is Input port and the input is sample based.

Interpolation

Input Dimensions (Sample Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Sample Based)
N-D Array (ex. M-by-N-by-Q)	Sample-based 1-by-P row	Applies I_{Pts} to the first dimension of the input array	P-by-N-by-Q array
	Sample- or frame-based P-by-1 column		
	Sample-based P-by-N-by-Q array	Applies the columns of I_{Pts} to the corresponding columns of the input array	
M-by-1 column	Sample-based 1-by-P row	Applies I_{Pts} to the input column	P-by-1 column
	Sample- or frame-based P-by-1 column		
1-by-N row	Sample-based 1-by-P row	Applies I_{Pts} to the input row	1-by-P row
	Sample or frame-based P-by-1 column		

The next table describes the block's behavior when the **Source of interpolation points** is Input port and the input is frame based.

Input Dimensions (Frame Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Frame Based)
M-by-N matrix	Frame-based 1-by-N row	Applies each column of I_{Pts} (each element of I_{Pts}) to the corresponding column of the input matrix	1-by-N row
	Sample-based 1-by-P row	Applies I_{Pts} to each input column	P-by-N matrix
	Frame- or sample-based P-by-1 column		
	Frame- or sample-based P-by-N matrix	Applies the columns of I_{Pts} to the corresponding columns of the input matrix	
M-by-1 column	Sample-based 1-by-P row	Applies I_{Pts} to the input column	P-by-1 column
	Frame- or sample-based P-by-1 column		

Interpolation

Input Dimensions (Frame Based)	Valid Dimensions of Interpolation Array I_{Pts}	How Block Applies I_{Pts} to Input	Output Dimensions (Frame Based)
1-by-N row (not recommended)	Frame-based 1-by-N row	Not Applicable. Block copies input vector	1-by-N row, a copy of the input vector
	Sample-based 1-by-P row		P-by-N matrix where each row is a copy of the input vector
	Frame- or sample-based P-by-1 column		
	Frame- or sample-based P-by-N matrix		

Handling Out-of-Range Interpolation Points

Valid values in the interpolation array I_{Pts} range from 1 to the number of samples in each channel of the input. For instance, given a length-5 input vector D, all entries of I_{Pts} must range from 1 to 5. I_{Pts} cannot contain entries such as 7 or -9, since there is no 7th or -9th entry in D.

The **Out of range interpolation points** parameter sets how the block handles interpolation points that are fall outside the valid range, and has the following settings:

- Clip — The block replaces any out-of-range values in I_{Pts} with the closest value in the valid range (from 1 to the number of input samples), and then proceeds with computations using the clipped version of I_{Pts}.

- Clip and warn — In addition to Clip, the block issues a warning at the MATLAB command line every time clipping occurs.

- `Error` — When the block encounters an out-of-range value in I_{Pts}, the simulation stops, and the block issues an error at the MATLAB command line.

Example of Clipping

Suppose the block is set to clip out-of-range interpolation points, and gets the following input vector and interpolation points:

- `D = [11 22 33 44]'`

- I_{Pts} `= [10 2.6 -3]'`

Because *D* has four samples, valid interpolation points range from 1 to 4. The block clips the interpolation point 10 to 4 and the point -3 to 1, resulting in the clipped interpolation vector $I_{\text{PtsClipped}}$ = `[4 2.6 1]'`.

Linear Interpolation Mode

When **Interpolation Mode** is set to `Linear`, the block interpolates data values by assuming that the data varies linearly between samples taken at adjacent sample times.

For instance, if the input signal *D* = [1 2 1.5 3 0.25]', the following plot on the left shows the samples in *D*, and the plot on the right shows the linearly interpolated values between the samples in *D*.
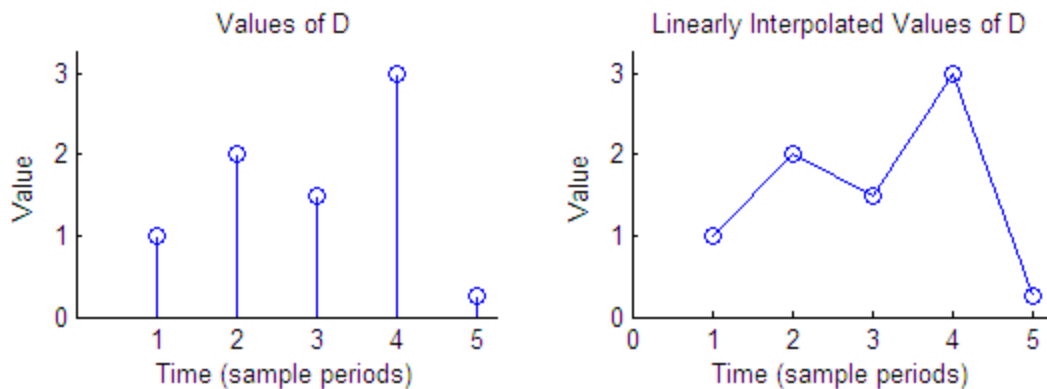
Interpolation

The following figure illustrates the case of a block in linear interpolation mode that is set to clip out-of-range interpolation points. The vector D supplies the input data and the vector I_{Pts} supplies the interpolation points:

- `D = [1 2 1.5 3 0.25]'`

- I_{Pts} `= [-4 2.7 4.3 10]'`

The block clips the invalid interpolation points, and outputs the linearly interpolated values in a vector, `[1 1.65 2.175 0.25]'`.

Interpolated Values of D at Clipped Interpolation Points
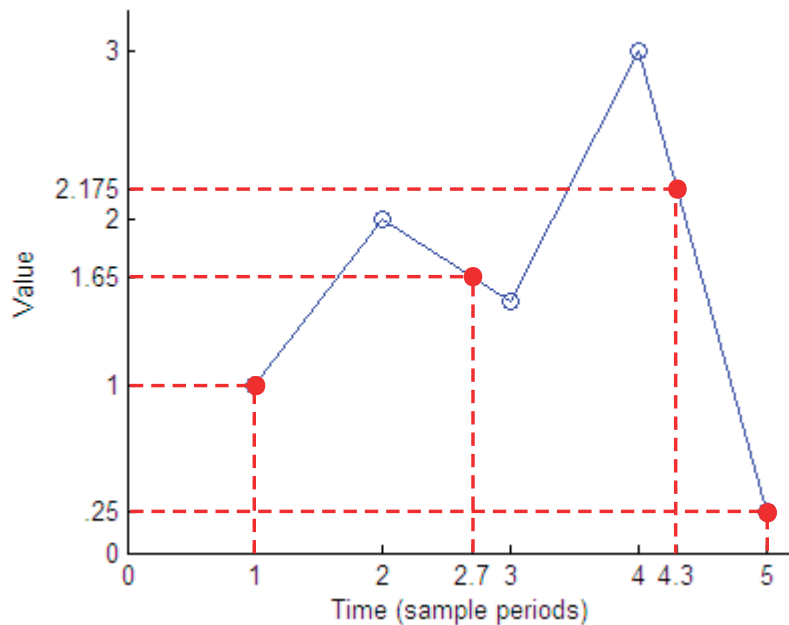
D = [1 2 1.5 3 0.25]'
Ipts = [-4 2.7 4.3 10]'

Valid interpolation points range from 1 to 5, so -4 is clipped to 1 and 10 is clipped to 5.

Clipped Ipts = [1 2.7 4.3 5]'

Output = [1 1.65 2.175 0.25]'

FIR Interpolation Mode

When **Interpolation Mode** is set to `FIR`, the block interpolates data values using an FIR interpolation filter, specified by various block

parameters. See FIR Interpolation Mode on page 1646 in the Variable Fractional Delay block reference for more information.
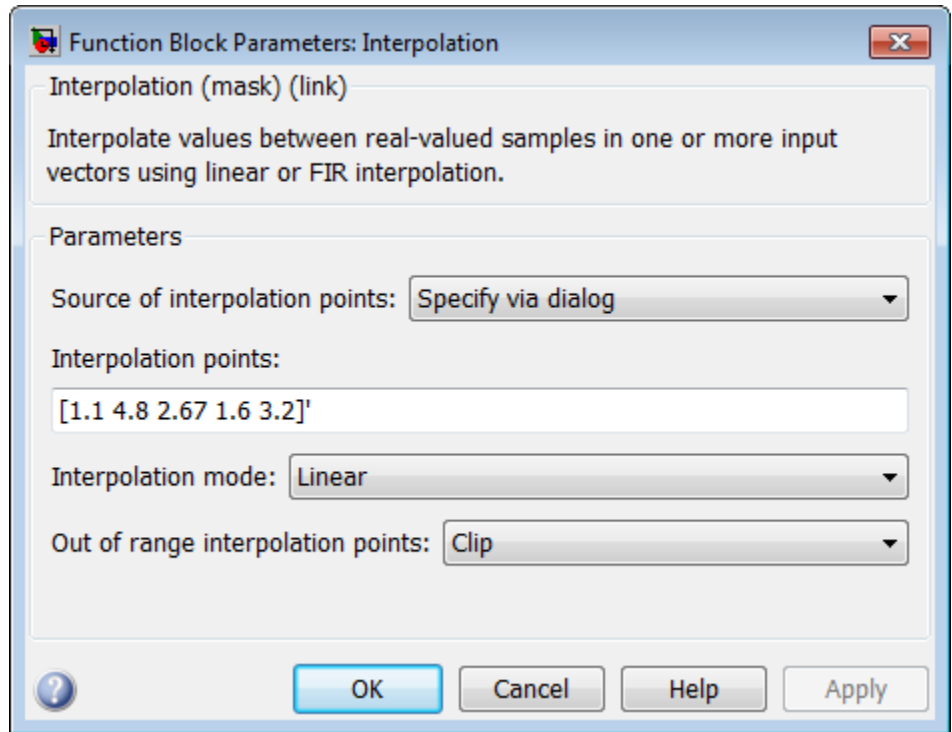
Dialog Box

Source of interpolation points

Choose how you want to specify the interpolation points. If you select Specify via dialog, the **Interpolation points** parameter become available. Use this option for static interpolation points. If you select Input port, the Pts port appears on the block. The block uses the input to this port as the interpolation points. Use this option for time-varying interpolation points. For more information, see "Specifying Static Interpolation Points" on page 2-814 and "Specifying Time-Varying Interpolation Points" on page 2-814.

Interpolation

Interpolation points

The array of points in time at which to interpolate the input signal (I_{Pts}). An entry of 1 in I_{Pts} refers to the first sample of the input, an entry of 2.5 refers to the sample half-way between the second and third input sample, and so on. See "How the Block Applies Interpolation Arrays to Inputs" on page 2-814. Tunable.

Interpolation mode

Sets the block to interpolate by either linear or FIR interpolation. For more information, see "Linear Interpolation Mode" on page 2-821 and "FIR Interpolation Mode" on page 2-822.

Interpolation filter half-length

Specify the half-length of the FIR interpolation filter (P). To perform the interpolation in FIR mode, the block uses the nearest 2*P low-rate samples. In most cases, P low-rate samples must appear below and above each interpolation point. However, if you interpolate at a low-rate sample point, the block includes that low-rate sample in the required 2*P samples and requires only 2*P–1 neighboring low-rate samples. If an interpolation point does not have the required number of neighboring low-rate samples, the block interpolates that point using linear interpolation.

This parameter becomes available only when the **Interpolation mode** is set to FIR. For more information, see "FIR Interpolation Mode" on page 2-822.

Interpolation points per input sample

Also known as the *upsampling factor*, this parameter defines the number of points per input sample (L) at which the block computes a unique FIR interpolation filter. To perform the FIR Interpolation, the block uses a polyphase structure with L filter arms of length 2*P.

For example, if L=4, the block constructs a polyphase filter with four arms. The block then interpolates at points corresponding to 1 +*i*/L, 2 +*i*/L, 3 +*i*/L..., where the integers 1, 2, and 3 represent the low-rate samples, and *i*=0,1,2,3. To interpolate at a point

that does not directly correspond to an arm of the polyphase filter requires an extra computation. The block first rounds that point down to the nearest value that does correspond to an arm of the polyphase filter. Thus, to interpolate at the point 2.2, the block rounds 2.2 down to 2, and computes the FIR interpolation using the first arm of the polyphase filter structure. Similarly, to interpolate the point 2.65, the block rounds the value down to 2.5 and uses the third arm of the polyphase filter structure.

This parameter becomes available only when the **Interpolation mode** is set to FIR. For more information, see "FIR Interpolation Mode" on page 2-822.

Normalized input bandwidth

The bandwidth of the input divided by Fs/2 (half the input sample frequency).

This parameter is only available when the **Interpolation mode** is set to FIR. For more information, see "FIR Interpolation Mode" on page 2-822.

Out of range interpolation points

When an interpolation point is out of range, this parameter sets the block to either clip the interpolation point, clip the value and issue a warning at the MATLAB command line, or stop the simulation and issue an error at the MATLAB command line. For more information, see "Handling Out-of-Range Interpolation Points" on page 2-820.

Interpolation

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point
Pts	• Double-precision floating point • Single-precision floating point
Out	• Double-precision floating point • Single-precision floating point

Purpose   Recover time-domain signals by performing inverse short-time, fast Fourier transform (FFT)

Library   Transforms

dspxfrm3

Description

The Inverse Short-Time FFT block reconstructs the time-domain signal from the frequency-domain output of the Short-Time FFT block using a two-step process. First, the block performs the overlap add algorithm shown below.

$$x[n] = \frac{L}{W(0)} \sum_{p=-\infty}^{\infty} \left[\frac{1}{N} \sum_{k=0}^{N-1} X[pL,k] e^{j2\pi kn/N} \right]$$

Then, the block rebuffers the signal in order to reconstruct the time-domain signal. Depending on the analysis window used by the Short-Time FFT block, the Inverse Short-Time FFT block might or might not achieve perfect reconstruction of the time domain signal.

Connect your complex-valued, single-channel or multichannel input signal to the X(n,k) port. The block accepts unoriented vector, column vector and matrix input. The block outputs the real or complex-valued, single-channel or multichannel inverse short-time FFT at port x(n).

Connect your complex-valued, single-channel analysis window to the w(n) port. When you select the **Assert if analysis window does not support perfect signal reconstruction** check box, the block displays an error when the input signal cannot be perfectly reconstructed. The block uses the values you enter for the **Analysis window length (W)** and **Reconstruction error tolerance**, or maximum amount of allowable error in the reconstruction process, to determine if the signal can be perfectly reconstructed.

Examples   The dspstsa example illustrates how to use the Short-Time FFT and Inverse Short-Time FFT blocks to remove the background noise from a speech signal.

Inverse Short-Time FFT
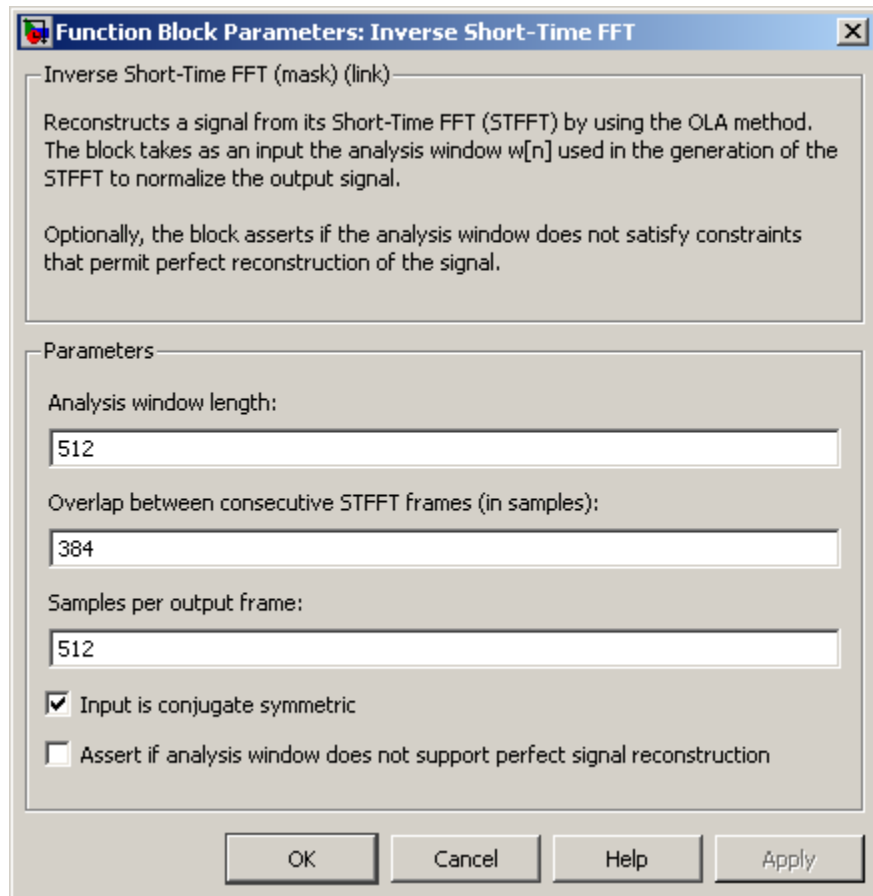
Dialog Box

Analysis window length

> Enter the length of the analysis window. This parameter is visible when you select the **Assert if analysis window does not support perfect signal reconstruction** check box.

Overlap between consecutive STFFT frames (in samples)

> Enter the number of samples of overlap for each frame of the Short-Time FFT block's input signal. This value should be the

same as the **Overlap between consecutive windows (in samples)** parameter in the Short-Time FFT block parameters dialog.

Samples per output frame
Enter the desired frame size of the output signal.

Input is conjugate symmetric
Select this check box when the input to the block is both floating point and conjugate symmetric, and you want real-valued outputs. When you select this check box when the input is not conjugate symmetric, the output of the block is invalid. This parameter cannot be used for fixed-point signals.

Assert if analysis window does not support perfect signal reconstruction
Select this check box to display an error when the analysis window used by the Short-Time FFT block does not support perfect signal reconstruction.

Reconstruction error tolerance
Enter the amount of acceptable error in the reconstruction of the original signal. This parameter is visible when you select the **Assert if analysis window does not support perfect signal reconstruction** check box.

References Quatieri, Thomas E. *Discrete-Time Speech Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 2001.

Inverse Short-Time FFT

Supported Data Types

Port	Supported Data Types
X(n,k)	• Double-precision floating point • Single-precision floating point
w(n)	• Double-precision floating point • Single-precision floating point
x(n)	• Double-precision floating point • Single-precision floating point

See Also

Burg Method	DSP System Toolbox
Magnitude FFT	DSP System Toolbox
Periodogram	DSP System Toolbox
Short-Time FFT	DSP System Toolbox
Spectrum Scope	DSP System Toolbox
Window Function	DSP System Toolbox
Yule-Walker Method	DSP System Toolbox
pwelch	Signal Processing Toolbox

Purpose   Design inverse sinc filter

Library   Filtering / Filter Designs

                   dspfdesign

Description   This block brings the filter design capabilities of the `filterbuilder`
                   function to the Simulink environment.
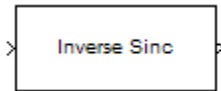
> Inverse Sinc >

Dialog   See "Inverse Sinc Filter Design Dialog Box — Main Pane" on page 7-735
Box   for more information about the parameters of this block. The **Data
                   Types** and **Code Generation** panes are not available for blocks in the
                   DSP System Toolbox Filter Designs library.

Inverse Sinc Filter

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Response type

Select `Lowpass` or `Highpass` to design an inverse sinc lowpass or highpass filter.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

Order

Enter the filter order. This option is enabled only if you set the **Order mode** to `Specify`.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Regions between specification values such as **Fpass** and **Fstop** represent transition regions where the filter response is not constrained.

Frequency constraints

When **Order mode** is `Specify`, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.

- `Passband edges` — For IIR filters, define the filter by specifying frequencies for the edges of the passband.

- `Stopband edges` — For IIR filters, define the filter by specifying frequencies for the edges of the stopbands.

- `6 dB point` — For FIR filters, define the filter response by specifying the locations of the 6 dB point. The 6 dB point is the frequency for the point six decibels below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

F6dB

When **Frequency constraints** is `6 dB point`, specify the frequency of the 6 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.

- `dB` — Specify the magnitude in decibels (default)

- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default FIR method is `Equiripple`.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options;

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set **Stopband shape**, **Stopband decay** specifies the amount of decay applied to the stopband. the following

conditions apply to **Stopband decay** based on the value of **Stopband Shape**:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. The block applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Sinc frequency factor
>   A frequency dilation factor. The **Sinc frequency factor**, C, parameterizes the passband magnitude response for a lowpass design through $H(\omega) = \text{sinc}(C\omega)^{\wedge}(-P)$ and through $H(\omega) = \text{sinc}(C(1-\omega))^{\wedge}(-P)$ for a highpass design.

Sinc power
>   Negative power of passband magnitude response. The **Sinc power**, P, parameterizes the passband magnitude response for a lowpass design through $H(\omega) = \text{sinc}(C\omega)^{\wedge}(-P)$ and through $H(\omega) = \text{sinc}(C(1-\omega))^{\wedge}(-P)$ for a highpass design.

Filter Implementation

Structure
>   For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use basic elements to enable filter customization
>   Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Note The Inherited (this choice will be removed   see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Inverse Sinc Filter

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.

- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Purpose   Compute filter estimates for inputs using Kalman adaptive filter
               algorithm

Library   dspobslib

Description

Note The Kalman Adaptive Filter block is still supported but is likely
to be obsoleted in a future release. We strongly recommend replacing
this block with the Kalman Filter block.

The Kalman Adaptive Filter block computes the optimal linear
minimum mean-square estimate (MMSE) of the FIR filter coefficients
using a one-step predictor algorithm. This Kalman filter algorithm is
based on the following physical realization of a dynamic system.
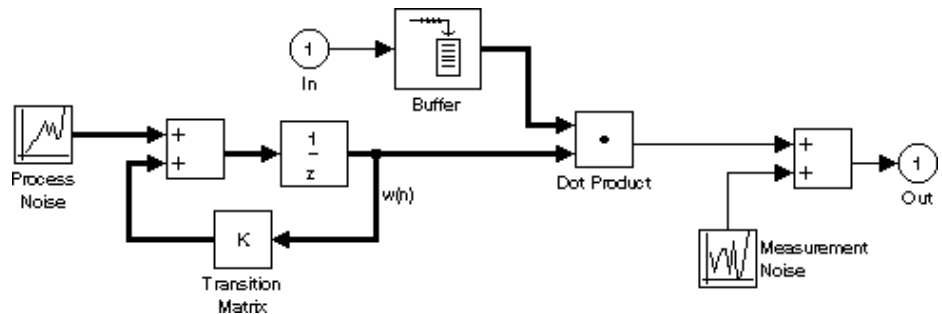
The Kalman filter assumes that there are no deterministic changes to
the filter taps over time (that is, the transition matrix is identity), and
that the only observable output from the system is the filter output
with additive noise. The corresponding Kalman filter is expressed in
matrix form as

Kalman Adaptive Filter (Obsolete)

$$g(n) = \frac{K(n-1)u(n)}{u^H(n)K(n-1)u(n) + Q_M}$$

$$y(n) = u^H(n)\hat{w}(n)$$

$$e(n) = d(n) - y(n)$$

$$\hat{w}(n+1) = \hat{w}(n) + e(n)g(n)$$

$$K(n) = K(n-1) - g(n)u^H(n)K(n-1) + Q_p$$

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 by scalar variance terms to be placed along the matrix diagonals, thus ensuring that these matrices are symmetric. The filter algorithm based on this constraint is also known as the random-walk Kalman filter.

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the input covariance matrix $K(n)$. This decreases the total number of computations by a factor of two.

The block icon has port labels corresponding to the inputs and outputs of the Kalman algorithm. Note that inputs to the In and Err ports must be sample-based scalars with the same complexity. The signal at the Out port is a scalar, while the signal at the Taps port is a sample-based vector.

Block Ports	Corresponding Variables
In	u, the scalar input, which is internally buffered into the vector $u(n)$
Out	$y(n)$, the filtered scalar output
Err	$e(n)$, the scalar estimation error
Taps	$\hat{w}(n)$, the vector of filter-tap estimates

An optional Adapt input port is added when you select the **Adapt port** check box in the dialog. When this port is enabled, the block continuously adapts the filter coefficients while the Adapt input is nonzero. A zero-valued input to the Adapt port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero Adapt input.
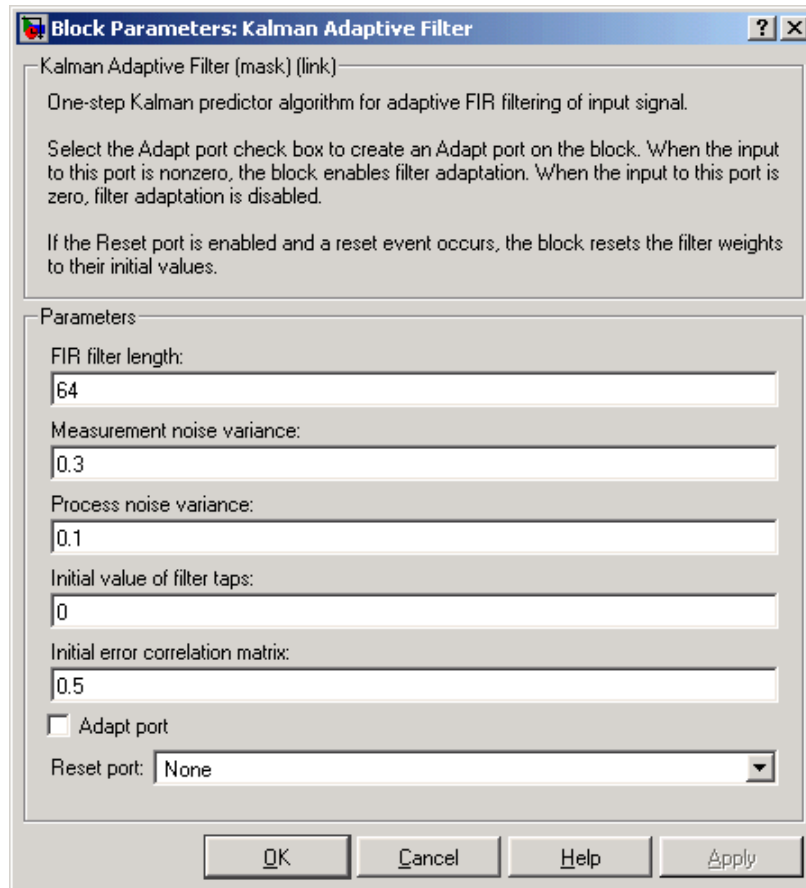
The **FIR filter length** parameter specifies the length of the filter that the Kalman algorithm estimates. The **Measurement noise variance** and the **Process noise variance** parameters specify the correlation matrices of the measurement and process noise, respectively. The **Measurement noise variance** must be a scalar, while the **Process noise variance** can be a vector of values to be placed along the diagonal, or a scalar to be repeated for the diagonal elements.

The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The **Initial**

error correlation matrix specifies the initial value *K(0)*, and can be a diagonal matrix, a vector of values to be placed along the diagonal, or a scalar to be repeated for the diagonal elements.

**Dialog
Box**

FIR filter length
    The length of the FIR filter.

Measurement noise variance
The value to appear along the diagonal of the measurement noise correlation matrix. Tunable.

Process noise variance
The value to appear along the diagonal of the process noise correlation matrix. Tunable.

Initial value of filter taps
The initial FIR filter coefficients.

Initial error correlation matrix
The initial value of the error correlation matrix.

Adapt port
Enables the Adapt port.

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

Supported Data Types

- Double-precision floating point

- Single-precision floating point

See Also

LMS Adaptive Filter (Obsolete)	DSP System Toolbox
RLS Adaptive Filter (Obsolete)	DSP System Toolbox

See "Adaptive Filters in Simulink" for related information.

Kalman Filter

Purpose Predict or estimate states of dynamic systems

Library Filtering/Adaptive Filters

dspadpt3
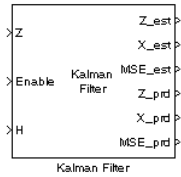
Description Use the Kalman Filter block to predict or estimate the state of a dynamic system from a series of incomplete and/or noisy measurements. Suppose you have a noisy linear system that is defined by the following equations:

$$x_k = Ax_{k-1} + w_{k-1}$$
$$z_k = Hx_k + v_k$$

This block can use the previously estimated state, $\hat{x}_{k-1}$, to predict the current state at time k, x_k^-, as shown by the following equation:

$$x_k^- = A\hat{x}_{k-1}$$
$$P_k^- = A\hat{P}_{k-1}A^T + Q$$

The block can also use the current measurement, z_k, and the predicted state, x_k^-, to estimate the current state value at time k, $\hat{x}_k$, so that it is a more accurate approximation:

$$K_k = P_k^- H^T \left(HP_k^- H^T + R \right)^{-1}$$
$$\hat{x}_k = x_k^- + K_k \left(z_k - Hx_k^- \right)$$
$$\hat{P}_k = \left(I - K_k H \right) P_k^-$$

The variables in the previous equations are defined in the following table.

Variable	Definition	Default Value or Initial Condition
x	State	N/A
$\hat{x}$	Estimated state	`zeros([6, 1])`
x^-	Predicted state	N/A
A	State transition matrix	$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$
w	Process noise	N/A
z	Measurement	N/A
H	Measurement matrix	$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$
v	Measurement noise	N/A
$\hat{P}$	Estimated error covariance	`10*eye(6)`
P^-	Predicted error covariance	N/A
Q	Process noise covariance	`0.05*eye(6)`
K	Kalman gain	N/A

Kalman Filter

Variable	Definition	Default Value or Initial Condition
R	Measurement noise covariance	`eye(4)`
I	Identity matrix	N/A

In the previous equations, *z* is a vector of measurement values. Most of the time, the block processes Z, an M-by-N matrix, where M is the number of measurement values and N is the number of filters.

Use the **Number of filters** parameter to specify the number of filters to use to predict or estimate the current value.

Use the **Enable filters** parameter to specify which filters are enabled or disabled at each time step. If you select `Always`, the filters are always enabled. If you choose `Specify via input port <Enable>`, the Enable port appears on the block. The input to this port must be a row vector of 1s and 0s whose length is equal to the number of filters. For example, if there are 3 filters and the input to the Enable port is [1 0 1], only the first and third filter are enabled at this time step. If you select the **Reset the estimated state and estimated error covariance when filters are disabled** check box, the estimated and predicted states as well as the estimated error covariance that correspond to the disabled filters are reset to their initial values.

Note All filters have the same state transition matrix, measurement matrix, initial conditions, and noise covariance, but their state, measurement, enable, and MSE signals are unique. Within the state, measurement, enable, and MSE signals, each column corresponds to a filter.

Use the **Measurement matrix source** parameter to specify how to enter the measurement matrix values. If you select `Specify via dialog`, the **Measurement matrix** parameter appears in the dialog

box. If you select `Input port <H>`, the H port appears on the block. Use this port to specify your measurement matrix.

See the Radar Tracking example for a demonstration of how to use this block. You can open this example by typing

`aero_radmod_dsp`

at the MATLAB command prompt.

Kalman Filter

The Kalman Filter dialog box appears as shown in the following figure.
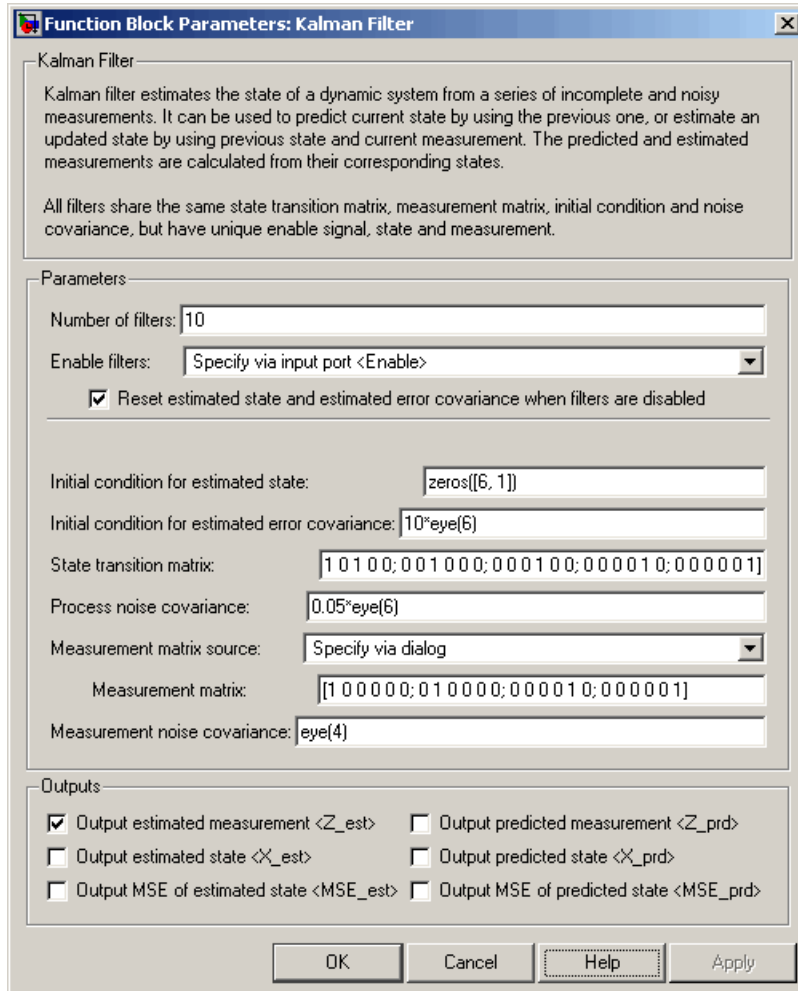
Number of filters

Specify the number of filters to use to predict or estimate the current value.

Enable filters

Specify which filters are enabled or disabled at each time step. If you select `Always`, the filters are always enabled. If you choose `Specify via input port <Enable>`, the Enable port appears on the block.

Reset the estimated state and estimated error covariance when filters are disabled

If you select this check box, the estimated and predicted states as well as the estimated error covariance that correspond to the disabled filters are reset to their initial values. This parameter is visible if, for the **Enable filters** parameter, you select `Specify via input port <Enable>`.

Initial condition for estimated state

Enter the initial condition for the estimated state.

Initial condition for estimated error covariance

Enter the initial condition for the estimated error covariance.

State transition matrix

Enter the state transition matrix.

Process noise covariance

Enter the process noise covariance.

Measurement matrix source

Specify how to enter the measurement matrix values. If you select `Specify via dialog`, the **Measurement matrix** parameter appears in the dialog box. If you select `Input port <H>`, the H port appears on the block.

Measurement matrix

Enter the measurement matrix values. This parameter is visible if you select `Specify via dialog` for the **Measurement matrix source** parameter.

Measurement noise covariance

Enter the measurement noise covariance.

Output estimated measurement <Z_est>
> Select this check box if you want the block to output the estimated measurement.

Output estimated state <X_est>
> Select this check box if you want the block to output the estimated state.

Output MSE of estimated state <MSE_est>
> Select this check box if you want the block to output the mean-squared error of the estimated state.

Output predicted measurement <Z_prd>
> Select this check box if you want the block to output the predicted measurement.

Output predicted state <X_prd>
> Select this check box if you want the block to output the predicted state.

Output MSE of predicted state <MSE_prb>
> Select this check box if you want the block to output the mean-squared error of the predicted state.

References

[1] Haykin, Simon. *Adaptive Filter Theory*. Upper Saddle River, NJ: Prentice Hall, 1996.

[2] Welch, Greg and Gary Bishop, "An Introduction to the Kalman Filter," TR 95–041, Department of Computer Science, University of North Carolina.

Supported Data Types

Port	Input/Output	Supported Data Types
Z	M-by-N measurement where M is the length of the measurement vector and N is the number of filters.	• Double-precision floating point • Single-precision floating point
Enable	1-by-N vector of 1s and 0s where N is the number of filters.	• Double-precision floating point • Single-precision floating point • Boolean
H	M-by-P measurement matrix where M is the length of the measurement vector and P is the length of the filter state vectors.	Same as Z port
Z_est	M-by-N estimated measurement matrix where M is the length of the measurement vector and N is the number of filters.	Same as Z port
X_est	P-by-N estimated state matrix where P is the length of the filter state vectors and N is the number of filters.	Same as Z port

Kalman Filter

Port	Input/Output	Supported Data Types
MSE_est	1-by-N vector that represents the mean-squared-error of the estimated state. N is the number of filters.	Same as Z port
Z_prd	M-by-N predicted measurement matrix where M is the length of the measurement vector and N is the number of filters.	Same as Z port
X_prd	P-by-N predicted state matrix where P is the length of the filter state vectors and N is the number of filters.	Same as Z port
MSE_prd	1-by-N vector that represents the mean-squared-error of the predicted state. N is the number of filters.	Same as Z port

See Also LDL Solver   DSP System Toolbox

Purpose   Factor square Hermitian positive definite matrices into lower, upper, and diagonal components

Library   Math Functions / Matrices and Linear Algebra / Matrix Factorizations

dspfactors

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^*. 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 than 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.

LDL Factorization

Fixed-Point Data Types

The following diagram shows the data types used within the LDL Factorization block for fixed-point signals.
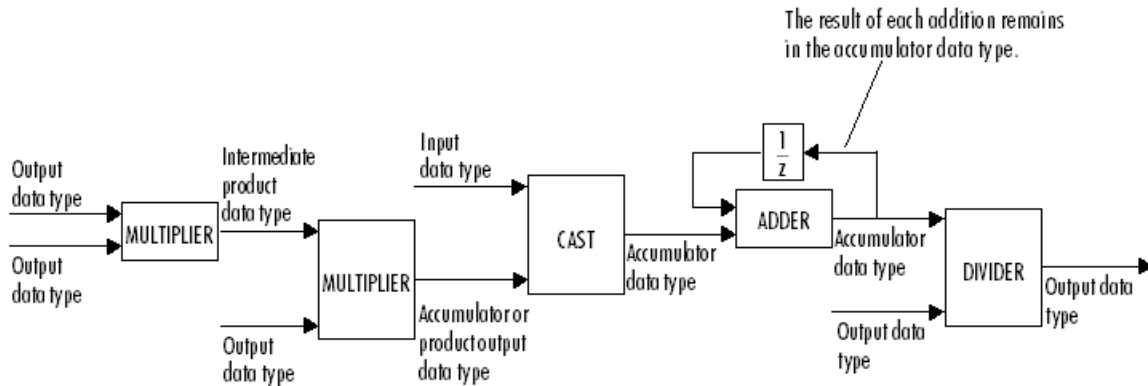
You can set the intermediate product, product output, accumulator, and output data types in the block dialog as discussed below.

The output of the second multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types".

Examples LDL decomposition of a 3-by-3 Hermitian positive definite matrix:

Dialog Box

The **Main** pane of the LDL Factorization block dialog appears as follows.
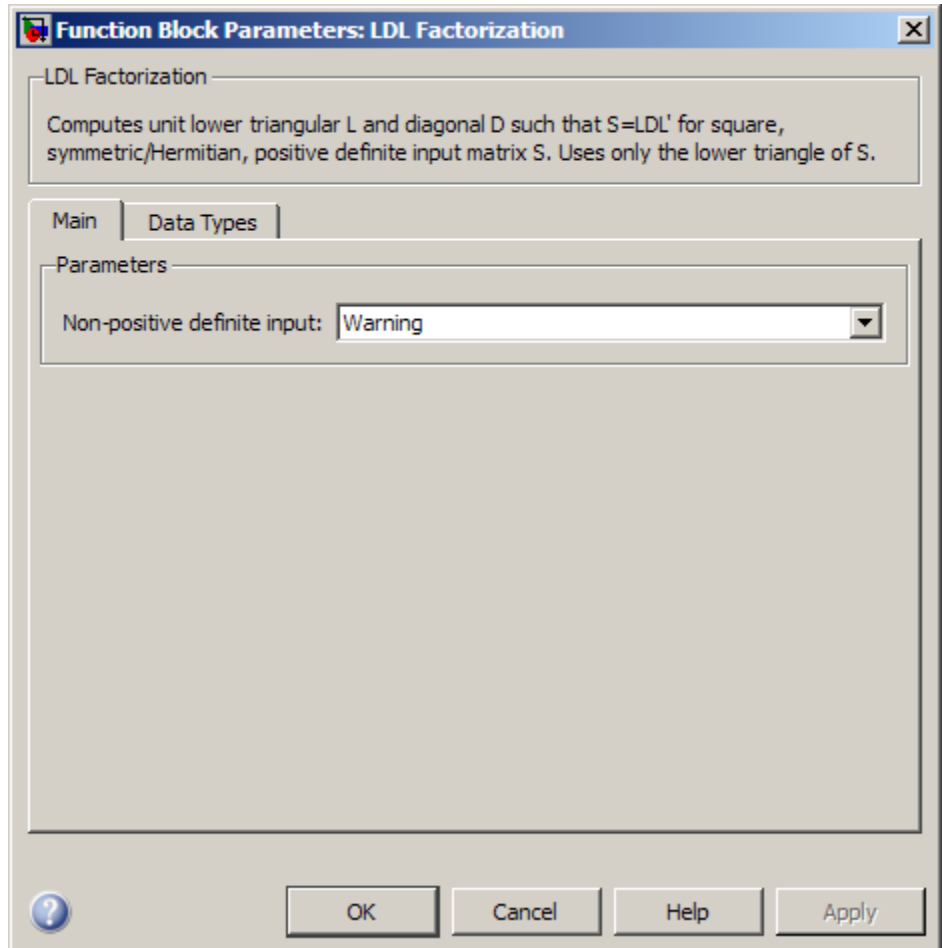
Non-positive definite input

Specify the action when nonpositive definite matrix inputs occur:

- `Ignore` — Proceed with the computation and do not issue an alert. The output is not a valid factorization. A partial factorization is 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 is present in the upper left corner of the output.

- `Error` — Display an error dialog and terminate the simulation.

The **Data Types** pane of the LDL Factorization block dialog appears as follows.
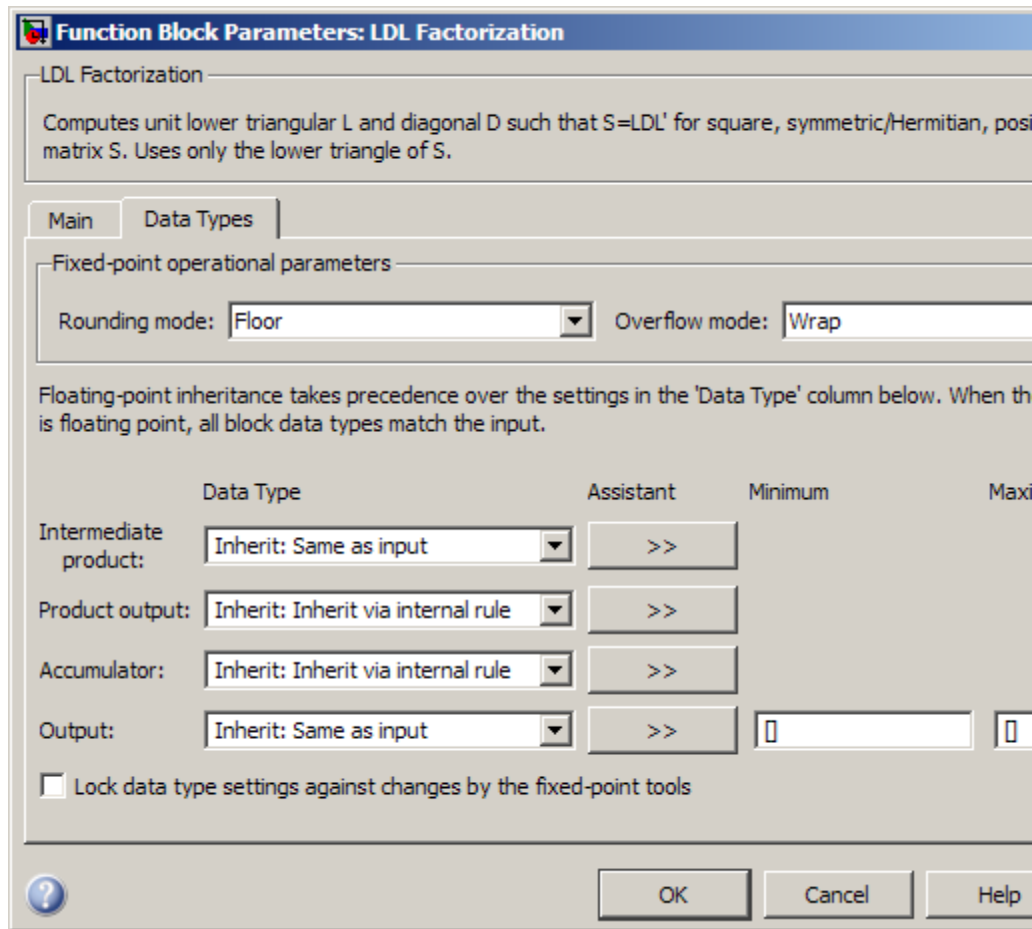
Rounding mode

Select the rounding mode for fixed-point operations.
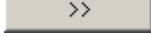
Overflow mode

Select the overflow mode for fixed-point operations.

Intermediate product

Specify the intermediate product data type. As shown in "Fixed-Point Data Types" on page 2-856, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. You can set it to:

- A rule that inherits a data type, for example, `Inherit:  Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [`>>`] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-856 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [`>>`] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-856 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-856 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

LDL Factorization

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
S	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
LDL'	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

See Also

Cholesky Factorization	DSP System Toolbox
LDL Inverse	DSP System Toolbox
LDL Solver	DSP System Toolbox
LU Factorization	DSP System Toolbox
QR Factorization	DSP System Toolbox

See "Matrix Factorizations" for related information.

Purpose   Compute inverse of Hermitian positive definite matrix using LDL factorization

Library   Math Functions / Matrices and Linear Algebra / Matrix Inverses

dspinverses
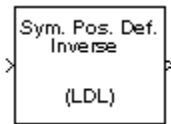
Description   The LDL Inverse block computes the inverse of the Hermitian positive definite input matrix S by performing an LDL factorization.

$$S^{-1} = \left(LDL^* \right)^{-1}$$

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.

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 and terminate the simulation.

LDL Inverse

> **Note** The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog, it is set to Ignore in the code generated for this block by Simulink Coder code generation software.
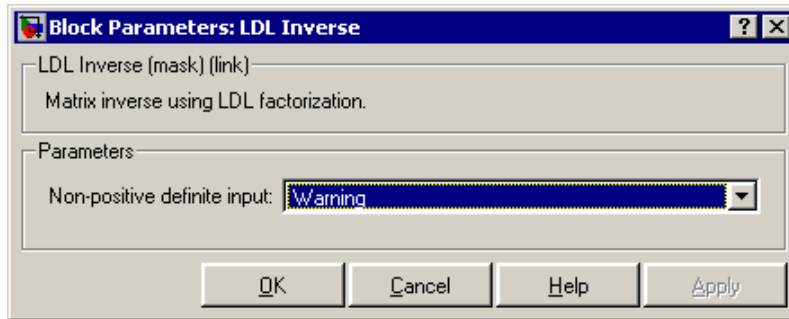
**Dialog
Box**

Non-positive definite input
   Response to nonpositive definite matrix inputs.

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

**Supported
Data
Types**

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

See Also

Cholesky Inverse	DSP System Toolbox
LDL Factorization	DSP System Toolbox
LDL Solver	DSP System Toolbox
LU Inverse	DSP System Toolbox

Pseudoinverse	DSP System Toolbox
inv	MATLAB

See "Matrix Inverses" for related information.

LDL Solver

Purpose   Solve *SX=B* for *X* when *S* is square Hermitian positive definite matrix

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers

dspsolvers
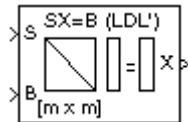
Description   The LDL Solver block solves the linear system *SX=B* by applying
LDL factorization to the matrix at the S port, which must be square
(*M*-by-*M*) and Hermitian positive definite. Only the diagonal and lower
triangle of the matrix are used, and any imaginary component of the
diagonal entries is disregarded. The input to the B port is the right side
M-by-*N* matrix, *B*. The *M*-by-*N* output matrix *X* is the unique solution
of the equations.

A length-*M* unoriented vector input for right side *B* is treated as an
M-by-1 matrix.

When the input is not positive definite, the block reacts with the
behavior specified by the **Non-positive definite input** parameter.
The following options are available:

- Ignore — Proceed with the computation and do not issue an alert.
  The output is not a valid solution.

- Warning — Proceed with the computation and display a warning
  message in the MATLAB Command Window. The output is not a
  valid solution.

- Error — Display an error dialog and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic
parameter. Like all diagnostic parameters on the Configuration
Parameters dialog, it is set to Ignore in the code generated for this
block by Simulink Coder code generation software.

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.

$$LY = B$$

$$DZ = Y$$

$$L^*X = Z$$

LDL Solver

Dialog Box

Non-positive definite input
> Response to nonpositive definite matrix inputs.

Supported Data Types

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

See Also

Autocorrelation LPC	DSP System Toolbox
Cholesky Solver	DSP System Toolbox
LDL Factorization	DSP System Toolbox
LDL Inverse	DSP System Toolbox
Levinson-Durbin	DSP System Toolbox
LU Solver	DSP System Toolbox
QR Solver	DSP System Toolbox

See "Linear System Solvers" for related information.

Purpose   Design and implement least-squares FIR filter

Library   dspobslib

Description
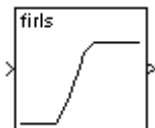
Note The Least Squares FIR Filter Design block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Least Squares FIR Filter Design block designs an FIR filter and applies it to a discrete-time input using the Direct Form II Transpose Filter block. The filter design uses the Signal Processing Toolbox `firls` function to minimize the integral of the squared error between the desired frequency response and the actual frequency response.

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

The **Filter type** parameter allows you to specify one of the following filters:

- Multiband — The Multiband filter designs a linear-phase filter with an arbitrary magnitude response.

- Differentiator — The Differentiator filter approximates the ideal differentiator. Differentiators are antisymmetric FIR filters with approximately linear magnitude responses. To obtain the correct derivative, scale the **Gains at these frequencies** vector by π F_s rad/s, where F_s is the sample frequency in Hertz.

- `Hilbert Transformer` — The `Hilbert Transformer` filter
  approximates the ideal Hilbert transformer. Hilbert transformers are
  antisymmetric FIR filters with approximately constant magnitude.

The **Band-edge frequency vector** parameter is a vector of frequency
points in the range 0 to 1, where 1 corresponds to half the sample
frequency. This vector must have even length, and intermediate points
must appear in ascending order. The **Gains at these frequencies**
parameter is a vector containing the desired magnitude response at the
corresponding points in the **Band-edge frequency vector**.

Each odd-indexed frequency-amplitude pair defines the left endpoint
of a line segment representing the desired magnitude response in that
frequency band. The corresponding even-indexed frequency-amplitude
pair defines the right endpoint. Between the frequency bands specified
by these end-points, there may be undefined sections of the specified
frequency response. These are called "don't care" or "transition"
regions, and the magnitude response in these areas is a result of the
optimization in the other (specified) frequency ranges.

The **Weights** parameter is a vector that specifies the emphasis to be placed on minimizing the error in certain frequency bands relative to others. This vector specifies one weight per band, so it is half the length of the **Band-edge frequency vector** and **Gains at these frequencies** vectors.

In most cases, differentiators and Hilbert transformers have only a single band, so the weight is a scalar value that does not affect the final filter. However, the **Weights** parameter is useful when using the block to design an antisymmetric multiband filter, such as a Hilbert transformer with stopbands.

For more information on the **Band-edge frequency vector**, **Gains at these frequencies**, and **Weights** parameters, see "Filter Designs and Implementation" in the Signal Processing Toolbox documentation. For more on the FIR filter algorithm, see the description of the `firls` function in the Signal Processing Toolbox documentation.

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`

Least Squares FIR Filter Design (Obsolete)

- **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

Least Squares FIR Filter Design (mask) (link)

Least-squares linear phase FIR filter.

Parameters

Filter type: Multiband

Filter order:
12

Band-edge frequency vector (including 0 and 1):
[0 0.3 0.6 1]

Gains at these frequencies:
[0 0 1 1]

Weights (one per band):
[1 1]

[OK] [Cancel] [Help] [Apply]
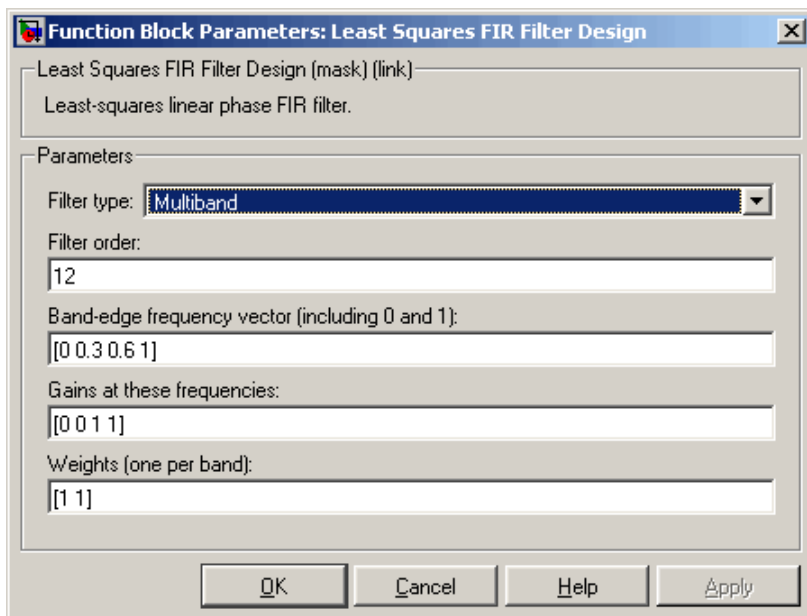
Filter type

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.

Least Squares FIR Filter Design (Obsolete)

Weights

A vector containing one weight for each frequency band. This vector must be half the length of the **Band-edge frequency vector** and **Gains at these frequencies** vectors. Tunable.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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

Purpose   Compute polynomial coefficients that best fit input data in least-squares sense

Library   Math Functions / Polynomial Functions

dsppolyfun

Description   The Least Squares Polynomial Fit block computes the coefficients of the nth order polynomial that best fits the input data in the least-squares sense, where you specify n in the **Polynomial order** parameter. A distinct set of $n+1$ coefficients is computed for each column of the M-by-N input, u.

For a given input column, the block computes the set of coefficients, c_1, c_2, ..., c_{n+1}, that minimizes the quantity

$$\sum_{i=1}^{M}(u_i - \hat{u}_i)^2$$

where u_i is the ith element in the input column, and

$$\hat{u}_i = f(x_i) = c_1 x_i^n + c_2 x_i^{n-1} + \; ... \; + c_{n+1}$$

The values of the independent variable, x_1, x_2, ..., x_M, are specified as a length-M vector by the **Control points** parameter. The same M control points are used for all N polynomial fits, and can be equally or unequally spaced. The equivalent MATLAB code is shown below.

```
c = polyfit(x,u,n)      % Equivalent MATLAB code
```

For convenience, the block treats length-M unoriented vector input as an M-by-1 matrix.

Each column of the $(n+1)$-by-N output matrix, c, represents a set of $n+1$ coefficients describing the best-fit polynomial for the corresponding column of the input. The coefficients in each column are arranged in order of descending exponents, c_1, c_2, ..., c_{n+1}.

Least Squares Polynomial Fit

Examples   In the ex_leastsquarespolyfit_ref 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.
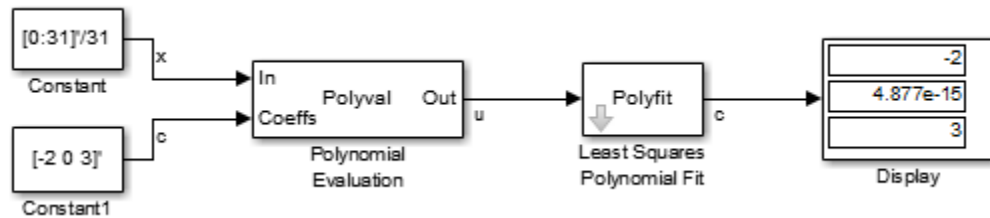
The **Control points** parameter of the Least Squares Polynomial Fit
block is configured with the same four values of independent variable u
that are used as input to the Polynomial Evaluation block, [1 2 3 4].
The Least Squares Polynomial Fit block uses these values together with
the input values of dependent variable y to reconstruct the original
polynomial coefficients.

Block Parameters: Least Squares Polynomial Fit

Polyfit (mask)

Find the coefficients of a polynomial P(X) of order N that fits the input data U, such that P(X) best approximates U in a least-squares sense. The input vector U must have the same length as X.

Parameters

Control points (X):

(0:31) / 31

Polynomial order (N):

3

| OK | Cancel | Help | Apply |
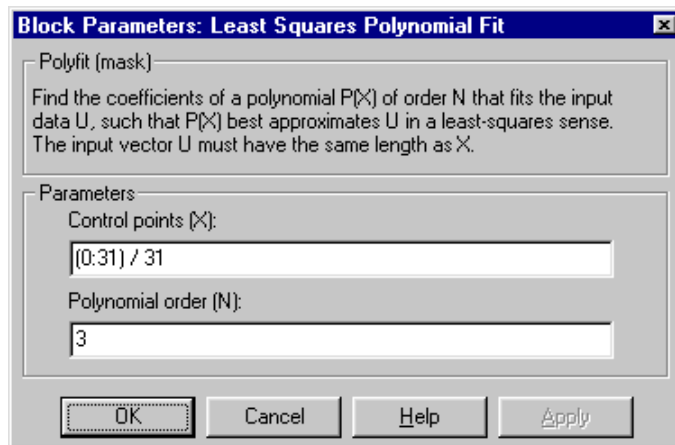
Dialog Box

Control points
The values of the independent variable to which the data in each input column correspond. For an *M*-by-*N* input, this parameter must be a length-*M* vector. Tunable.

Polynomial order
The order, *n*, of the polynomial to be used in constructing the best fit. The number of coefficients is *n*+1.

Supported Data Types
- Double-precision floating point
- Single-precision floating point

See Also

Detrend	DSP System Toolbox
Polynomial Evaluation	DSP System Toolbox
Polynomial Stability Test	DSP System Toolbox
polyfit	MATLAB

Levinson-Durbin

Purpose   Solve linear system of equations using Levinson-Durbin recursion

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers

dspsolvers

Description
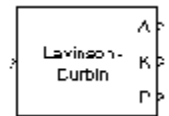
The Levinson-Durbin block solves the nth-order system of linear equations

$$Ra = b$$

in the cases where:

- R is a Hermitian, positive-definite, Toeplitz matrix.

- b is identical to the first column of R shifted by one element and with the opposite sign.

$$\begin{bmatrix} r(1) & r^*(2) & \cdots & r^*(n) \\ r(2) & r(1) & \cdots & r^*(n-1) \\ \vdots & \vdots & \ddots & \vdots \\ r(n) & r(n-1) & \cdots & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(n+1) \end{bmatrix}$$

The input to the block, r = [r(1) r(2) ... r(n+1)], can be a vector or a matrix. If the input is a matrix, the block treats each column as an independent channel and solves it separately. Each channel of the input contains lags 0 through n of an autocorrelation sequence, which appear in the matrix R.

The block can output the polynomial coefficients, A, the reflection coefficients, K, and the prediction error power, P, in various combinations. The **Output(s)** parameter allows you to enable the A and K outputs by selecting one of the following settings:

- A — For each channel, port A outputs A=[1 a(2) a(3) ... a(n+1)], the solution to the Levinson-Durbin equation. A has the same dimension as the input. You can also view the elements of each

output channel as the coefficients of an nth-order autoregressive (AR) process.

- K — For each channel, port K outputs $K=[k(1) \quad k(2) \quad \ldots \quad k(n)]$, which contains n reflection coefficients and has the same dimension as the input, less one element. A scalar input channel causes an error when you select K. You can use reflection coefficients to realize a lattice representation of the AR process described later in this page.

- A and K — The block outputs both representations at their respective ports. A scalar input channel causes an error when you select A and K.

Select the **Output prediction error power (P)** check box to output the prediction error power for each channel, *P*. For each channel, *P* represents the power of the output of an FIR filter with taps *A* and input autocorrelation described by *r*, where *A* represents a prediction error filter and *r* is the input to the block. In this case, *A* is a whitening filter. *P* has one element per input channel.

When you select the **If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0** check box (default), an input channel whose $r(1)$ element is zero generates a zero-valued output. When you clear this check box, an input with $r(1) = 0$ generates NaNs in the output. In general, an input with $r(1) = 0$ is invalid because it does not construct a positive-definite matrix *R*. Often, however, blocks receive zero-valued inputs at the start of a simulation. The check box allows you to avoid propagating NaNs during this period.

Applications

One application of the Levinson-Durbin formulation implemented by this block is in the Yule-Walker AR problem, which concerns modeling an unknown system as an autoregressive process. You would model such a process as the output of an all-pole IIR filter with white Gaussian noise input. In the Yule-Walker problem, the use of the signal's autocorrelation sequence to obtain an optimal estimate leads to an *Ra* = *b* equation of the type shown above, which is most efficiently solved by Levinson-Durbin recursion. In this case, the input to the block

represents the autocorrelation sequence, with `r(1)` being the zero-lag value. The output at the block's A port then contains the coefficients of the autoregressive process that optimally models the system. The coefficients are ordered in descending powers of z, and the AR process is minimum phase. The prediction error, G, defines the gain for the unknown system, where $G = \sqrt{P}$:

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ ... \ + a(n+1)z^{-n}}$$

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).

$$H(z) = A(z) = 1 + a(2)z^{-1} + \ ... \ a(n+1)z^{-n}$$

These coefficients solve the following optimization problem:

$$\min_{\{a_i\}}$$

$$E\left[\left| x_n - \sum_{i=1}^{N} a_i x_{n-i} \right|^2 \right]$$

Again, the output at the block's K port contains the corresponding reflection coefficients, [k(1) k(2) ... k(n)], for the lattice realization of this FIR filter. The Autocorrelation LPC block in the Linear Prediction library implements this autocorrelation-based prediction method.

Fixed-Point Data Types

The diagrams in this section show the data types used within the Levinson-Durbin block for fixed-point signals.

After initialization the block performs n updates. At the $(j+1)$ update,

$$\text{value in accumulator} = r(j+1) + \sum a_j(i) \times r(j-i+1)$$

The following diagram displays the fixed-point data types used in this calculation:
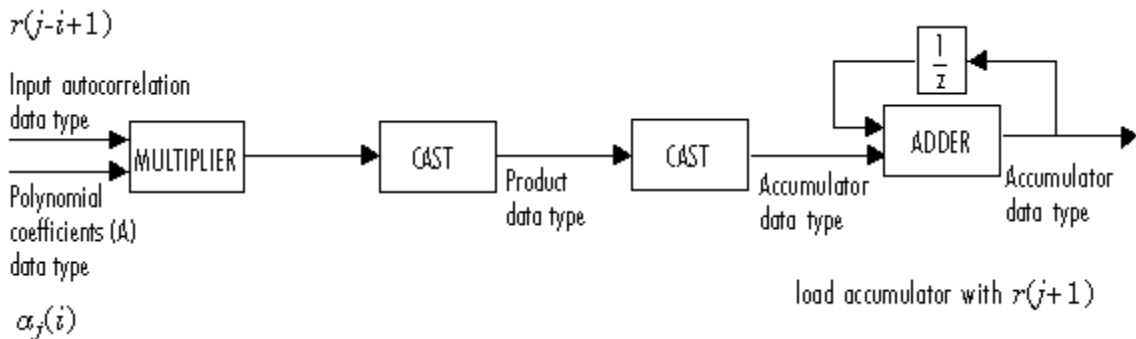
The block then updates the reflection coefficients K according to

$$K_{j+1} = \text{value in accumulator} / P_j$$

The block then updates the prediction error power P according to

$$P_{j+1} = P_j - P_j \times K_{j+1} \times \text{conj}(K_{j+1})$$

The next diagram displays the fixed-point data types used in this calculation:

$$r(j-i+1)$$

Input autocorrelation data type

Polynomial coefficients (A) data type

$$a_j(i)$$

load accumulator with $r(j+1)$
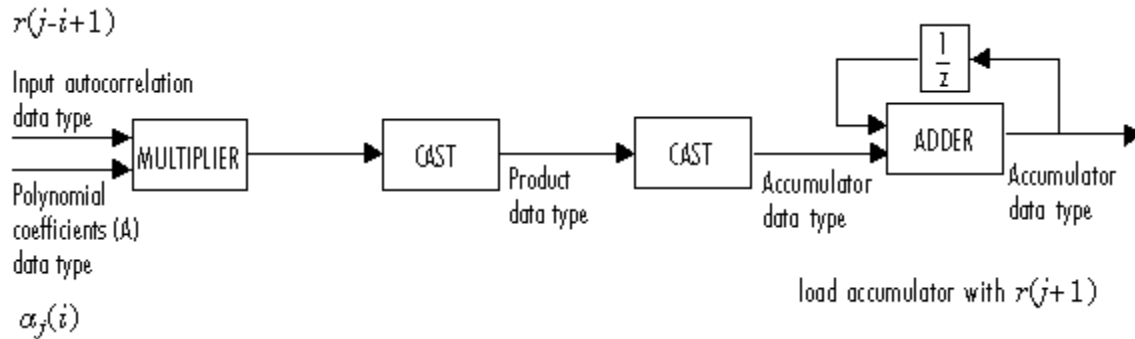
The polynomial coefficients *A* are then updated according to

$$a_{j+1}(i) = a_j(i) + K_{j+1} \times \mathrm{conj}\big(a_j(j-1+i)\big)$$

This diagram displays the fixed-point data types used in this calculation:

$$K_{j+1}$$

Reflection coefficients (K) data type

Reflection coefficients (K) data type

$$\mathrm{conj}(K_{j+1})$$

$$P_j$$

load accumulator with P_j

Prediction error power (P) data type

$$P_{j+1}$$
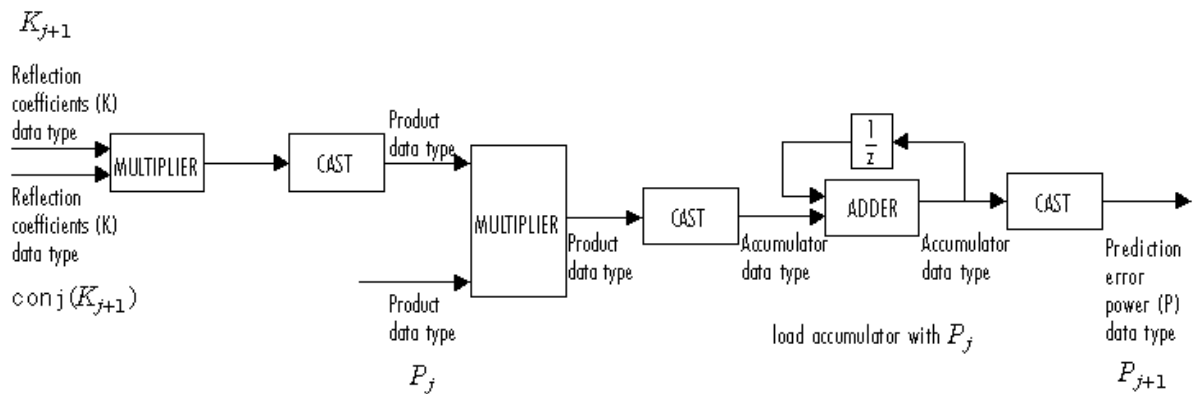
Algorithm   The algorithm requires $O(n^2)$ operations for each input channel. This implementation is therefore much more efficient for large *n* than

standard Gaussian elimination, which requires $O(n^3)$ operations per channel.

Dialog Box

The **Main** pane of the Levinson-Durbin block dialog box appears as follows.
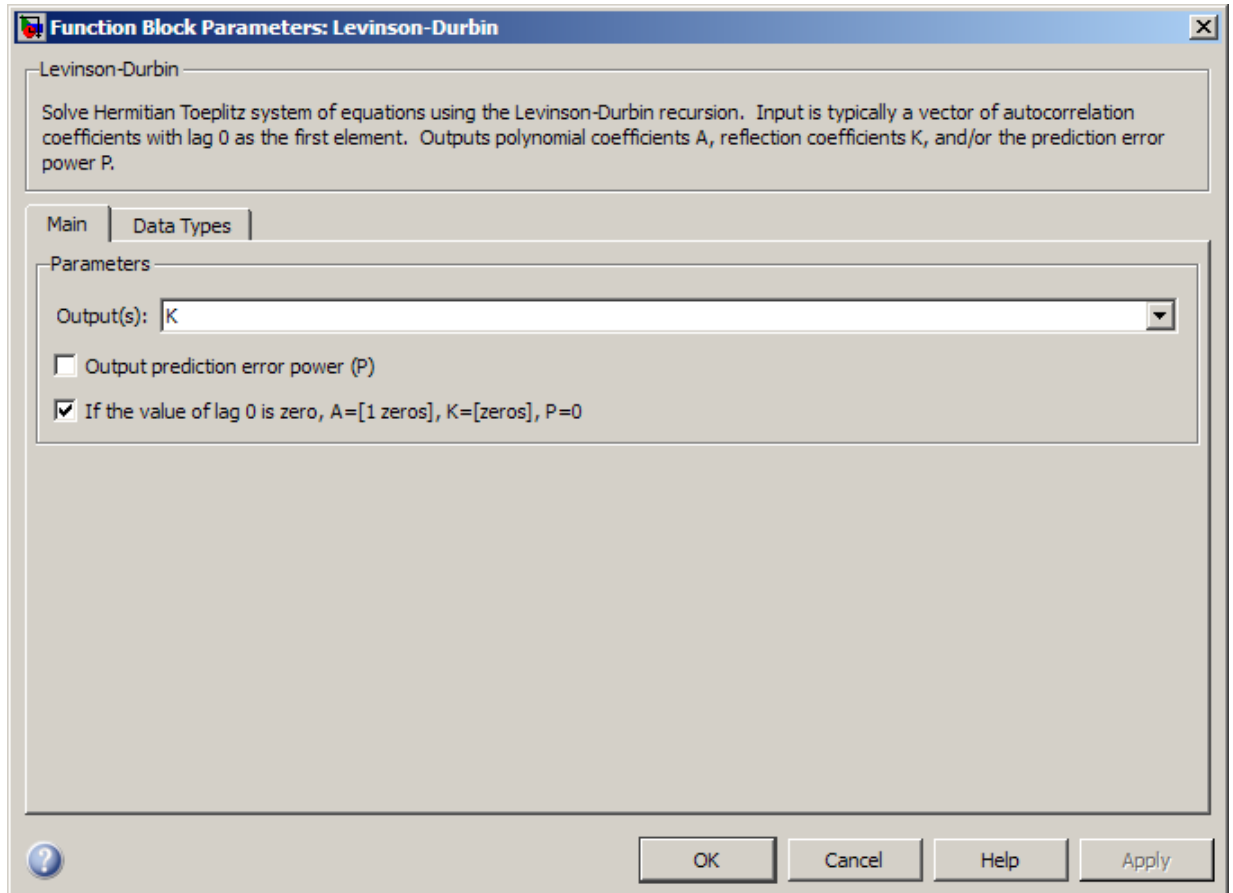
Output(s)

Specify the solution representation of $Ra = b$ to output: model coefficients (A), reflection coefficients (K), or both (A and K). When the input is a scalar or row vector, you must set this parameter to A.

Output prediction error power (P)

Select to output the prediction error at port P.

If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0

When you select this check box and the first element of the input, r(1), is zero, the block outputs the following vectors, as appropriate:

- A = [1 zeros(1,n)]

- K = [zeros(1,n)]

- P = 0

When you clear this check box, the block outputs a vector of NaNs for each channel whose r(1) element is zero.

The **Data Types** pane of the Levinson-Durbin block dialog box appears as follows.
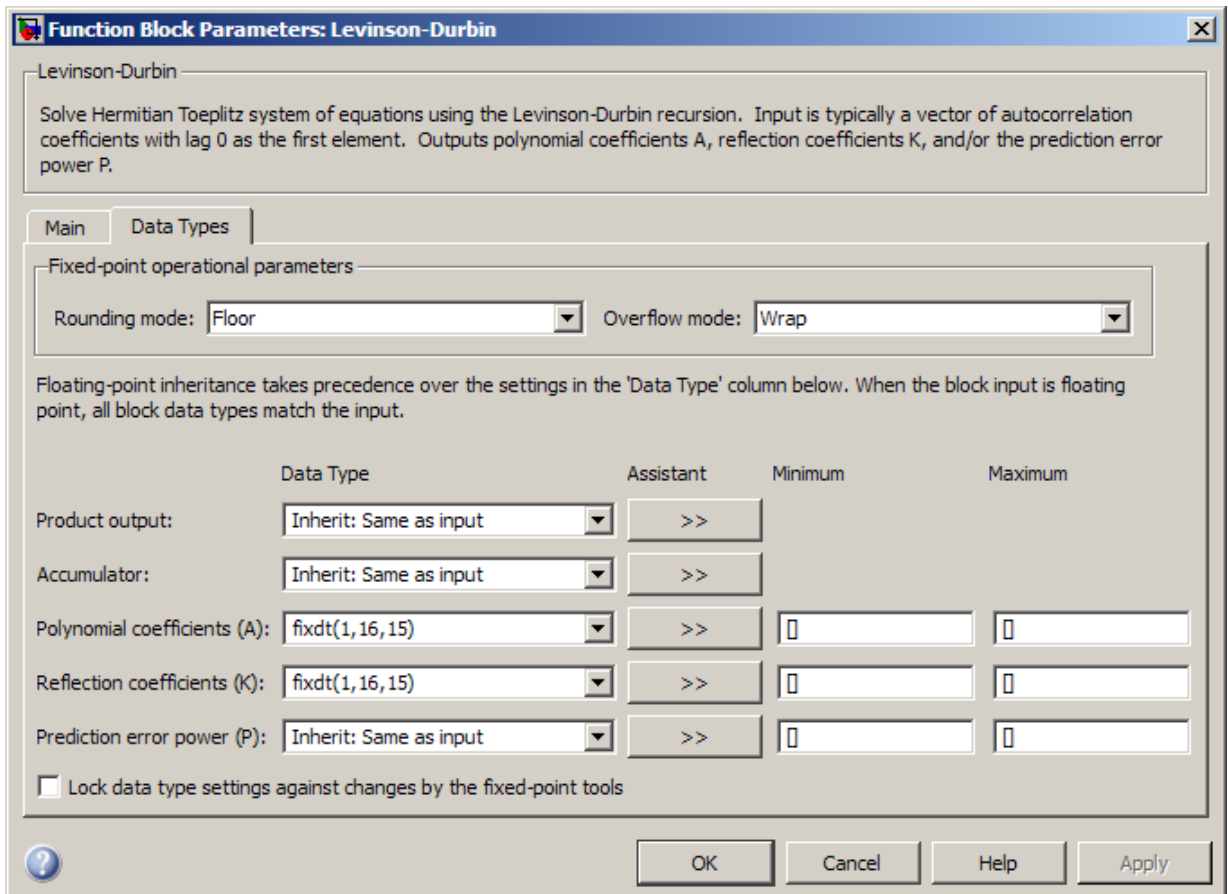
Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode
Select the rounding mode for fixed-point operations.
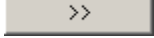
Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-881 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-881 for illustrations depicting the use of the accumulator data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Polynomial coefficients (A)

Specify the polynomial coefficients (*A*) data type. See "Fixed-Point Data Types" on page 2-881 for illustrations depicting the use of the A data type in this block. You can set it to an expression that evaluates to a valid data type, for example, `fixdt(1,16,15)`.

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **A** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Reflection coefficients (K)

Specify the polynomial coefficients (*A*) data type. See "Fixed-Point Data Types" on page 2-881 for illustrations depicting the use of the K data type in this block. You can set it to an expression that evaluates to a valid data type, for example, `fixdt(1,16,15)`.
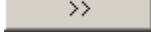
Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **K** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Prediction error power (P)

Specify the prediction error power (*P*) data type. See "Fixed-Point Data Types" on page 2-881 for illustrations depicting the use of the P data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

2-887

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **P** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Minimum
Specify the minimum values that the polynomial coefficients, reflection coefficients, or prediction error power should have. The default value is [ ] (unspecified).Simulink software uses this value to perform:

- Parameter range checking (see "Check Parameter Values")

- Automatic scaling of fixed-point data types

Maximum
Specify the maximum values that the polynomial coefficients, reflection coefficients, or prediction error power should have. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see "Check Parameter Values")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

References

Golub, G. H. and C. F. Van Loan. Sect. 4.7 in *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

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

Kay, Steven M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1988.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

Cholesky Solver	DSP System Toolbox
LDL Solver	DSP System Toolbox
Autocorrelation LPC	DSP System Toolbox
LU Solver	DSP System Toolbox
QR Solver	DSP System Toolbox
Yule-Walker AR Estimator	DSP System Toolbox
Yule-Walker Method	DSP System Toolbox
levinson	Signal Processing Toolbox

See "Linear System Solvers" for related information.

LMS Adaptive Filter (Obsolete)

Purpose   Compute filter estimates for input using LMS adaptive filter algorithm

Library   dspobslib

Description

In	Out
nLMS	
Err	Taps
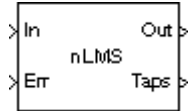
Note The LMS Adaptive Filter block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the LMS Filter block.

The LMS Adaptive Filter block implements an adaptive FIR filter using the stochastic gradient algorithm known as the normalized least mean-square (LMS) algorithm.

$$y(n) = \hat{w}^H(n-1)u(n)$$
$$e(n) = d(n) - y(n)$$
$$\hat{w}(n) = \hat{w}(n-1) + \frac{u(n)}{a + u^H(n)u(n)}\mu e^*(n)$$

The variables are as follows.

Variable	Description
n	The current algorithm iteration
$u(n)$	The buffered input samples at step n
$\hat{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
μ	The adaptation step size

To overcome potential numerical instability in the tap-weight update, a small positive constant (a = 1e-10) has been added in the denominator.

To turn off normalization, clear the **Use normalization** check box in the parameter dialog. 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

LMS Adaptive Filter (Obsolete)

An optional `Adapt` input port is added when you select the **Adapt input** check box in the dialog. When this port is enabled, the block continuously adapts the filter coefficients while the `Adapt` input is nonzero. A zero-valued input to the Adapt port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero `Adapt` input.

The **FIR filter length** parameter specifies the length of the filter that the LMS algorithm estimates. The **Step size** parameter corresponds to μ in the equations. Typically, for convergence in the mean square, μ must be greater than 0 and less than 2. The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The **Leakage factor** specifies the value of the leakage factor, $1 - \mu\,\alpha$, in the leaky LMS algorithm below. This parameter must be between 0 and 1.

$$\hat{w}(n+1) = (1 - \mu\alpha)\hat{w}(n) + \frac{u(n)}{u^H(n)u(n)}\mu e^*(n)$$

Examples See the `lmsadeq` and `lmsadtde` demos.

**Dialog
Box**
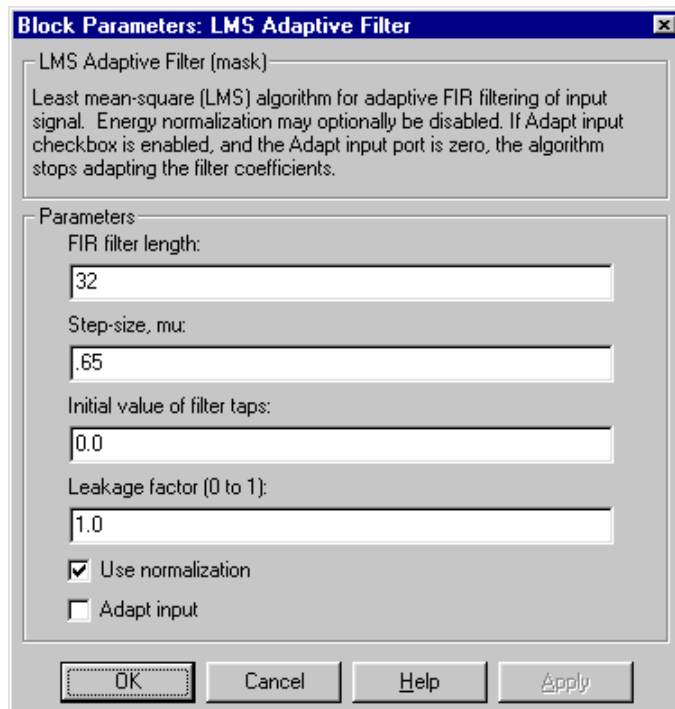
FIR filter length
The length of the FIR filter.

Step-size
The step-size, usually in the range (0, 2). Tunable.

Initial value of filter taps
The initial FIR filter coefficients.

Leakage factor
The leakage factor, in the range [0, 1]. Tunable.

Use normalization
Select this check box to compute the filter-tap estimate using the
normalized equations.

LMS Adaptive Filter (Obsolete)

Adapt input
> Enables the Adapt port when selected.

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

Supported Data Types
- Double-precision floating point
- Single-precision floating point

See Also

Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
RLS Adaptive Filter (Obsolete)	DSP System Toolbox

See "Adaptive Filters in Simulink" for related information.

Purpose   Compute output, error, and weights using LMS adaptive algorithm

Library   Filtering / Adaptive Filters

dspadpt3

Description
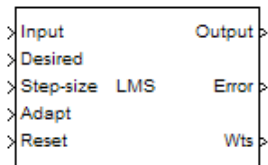
The LMS Filter block can implement an adaptive FIR filter using five different algorithms. The block estimates the filter weights, or coefficients, needed to minimize the error, *e(n)*, between the output signal *y(n)* and the desired signal, *d(n)*. Connect the signal you want to filter to the Input port. The input signal can be a scalar or a column vector. Connect the desired signal to the Desired port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal, which is the estimate of the desired signal. The Error port outputs the result of subtracting the output signal from the desired signal.

When you select LMS for the **Algorithm** parameter, the block calculates the filter weights using the least mean-square (LMS) algorithm. This algorithm is defined by the following equations.

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

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

$$f(\mathbf{u}(n), e(n), \mu) = \mu e(n)\mathbf{u}^*(n)$$

The variables are as follows.

Variable	Description
n	The current time index
$\mathbf{u}(n)$	The vector of buffered input samples at step n
$\mathbf{u^*}(n)$	The complex conjugate of the vector of buffered input samples at step n
$\mathbf{w}(n)$	The vector of filter weight estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
μ	The adaptation step size

When you select Normalized LMS for the **Algorithm** parameter, the block calculates the filter weights using the normalized LMS algorithm. The weight update function for the normalized LMS algorithm is defined as

$$f(\mathbf{u}(n), e(n), \mu) = \mu e(n) \frac{\mathbf{u}^*(n)}{\varepsilon + \mathbf{u}^H(n)\mathbf{u}(n)}$$

To overcome potential numerical instability in the update of the weights, a small positive constant, epsilon, has been added in the denominator. For double-precision floating-point input, epsilon is 2.2204460492503131e-016. For single-precision floating-point input, epsilon is 1.192092896e-07. For fixed-point input, epsilon is 0.

When you select Sign-Error LMS for the **Algorithm** parameter, the block calculates the filter weights using the LMS algorithm equations. However, each time the block updates the weights, it replaces the error

term *e(n)* with +1 when the error term is positive, -1 when it is negative, or 0 when it is zero.

When you select Sign-Data LMS for the **Algorithm** parameter, the block calculates the filter weights using the LMS algorithm equations. However, each time the block updates the weights, it replaces each sample of the input vector *u* (*n*) with +1 when the input sample is positive, -1 when it is negative, or 0 when it is zero.

When you select Sign-Sign LMS for the **Algorithm** parameter, the block calculates the filter weights using the LMS algorithm equations. However, each time the block updates the weights, it replaces the error term *e(n)* with +1 when the error term is positive, -1 when it is negative, or 0 when it is zero. It also replaces each sample of the input vector *u* (*n*) with +1 when the input sample is positive, -1 when it is negative, or 0 when it is zero.

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

The **Step size (mu)** parameter corresponds to μ in the equations. For convergence of the normalized LMS equations, $0<\mu<2$. You can either specify a step size using the input port, Step-size, or by entering a value in the Block Parameters: LMS Filter dialog.

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

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

When you select LMS from the **Algorithm** list, the weight update function in the above equation is the LMS weight update function. When you select Normalized LMS from the **Algorithm** list, the weight update function in the above equation is the normalized LMS weight update function.

Enter the initial filter weights $\mathbf{w}(0)$ as a vector or a scalar in the **Initial value of filter weights** text box. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector

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

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

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

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

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

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

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

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

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

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

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

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

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

Fixed-Point Data Types

The following diagrams show the data types used within the LMS Filter block for fixed-point signals; the table summarizes the definitions of variables used in the diagrams:

LMS Filter

Variable	Definition
$\mathbf{u}$	Input vector
$\mathbf{W}$	Vector of filter weights
μ	Step size
e	Error
Q	Quotient, $Q = \dfrac{\mu \cdot e}{\mathbf{u}'\mathbf{u}}$
Product $\mathbf{u}'\mathbf{u}$	Product data type in Energy calculation diagram
Accumulator $\mathbf{u}'\mathbf{u}$	Accumulator data type in Energy calculation diagram
Product $\mathbf{W}'\mathbf{u}$	Product data type in Convolution diagram
Accumulator $\mathbf{W}'\mathbf{u}$	Accumulator data type in Convolution diagram
Product $\mu \cdot e$	Product data type in Product of step size and error diagram
Product $Q \cdot \mathbf{u}$	Product and accumulator data type in Weight update diagram. [1]

[1]The accumulator data type for this quantity is automatically set to be the same as the product data type. The minimum, maximum, and overflow information for this accumulator is logged as part of the product information. Autoscaling treats this product and accumulator as one data type.

Energy calculation (for normalized LMS algorithm only)

input
data type

product (u'u)
data type

accumulator
(u'u) data type

accumulator
(u'u) data type

Convolution

input
data type

weight
data type

product
(W'u)
data type

accumu-
lator (W'u)
data type

accumu-
lator (W'u)
data type

output
data type

(same as
input signal
data type)

Output error signal

desired signal
data type
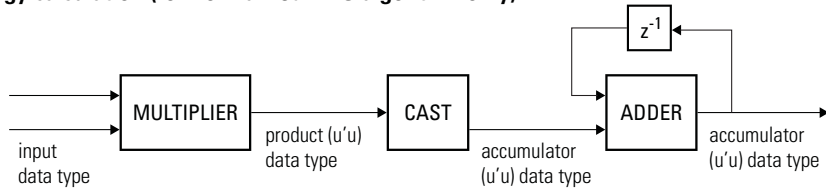
output signal
data type
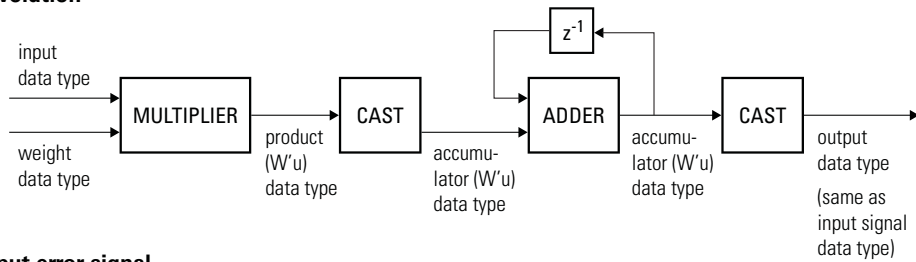
error signal
data type
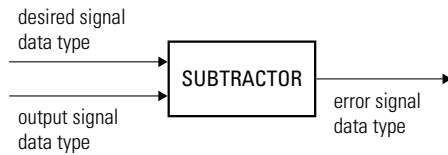
LMS Filter

Product of step size and error (for LMS and Sign-Data LMS algorithms only)
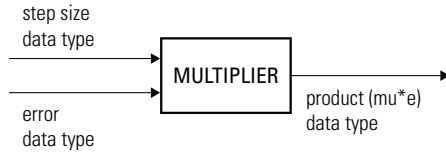
Quotient (for normalized LMS only)
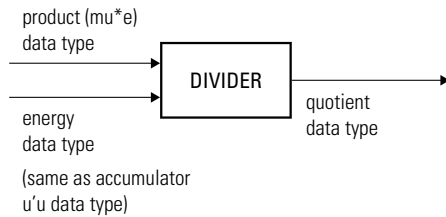
Weight update
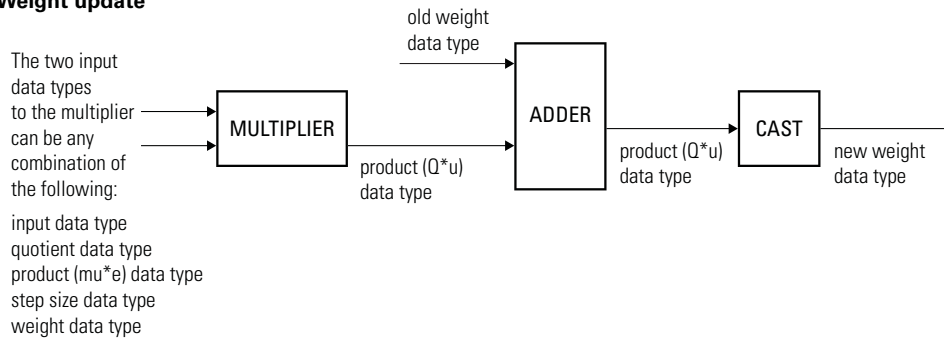
You can set the data type of the parameters, weights, products, quotient, and accumulators in the block mask. Fixed-point inputs, outputs, and mask parameters of this block must have the following characteristics:

- The input signal and the desired signal must have the same word length, but their fraction lengths can differ.

- The step size and leakage factor must have the same word length, but their fraction lengths can differ.

- The output signal and the error signal have the same word length and the same fraction length as the desired signal.

- The quotient and the product output of the **u'u**, **W'u**, $\mu \cdot e$, and $Q \cdot \mathbf{u}$ operations must have the same word length, but their fraction lengths can differ.

- The accumulator data type of the **u'u** and **W'u** operations must have the same word length, but their fraction lengths can differ.

The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types".

LMS Filter

Dialog Box

The **Main** pane of the LMS Filter block dialog appears as follows.
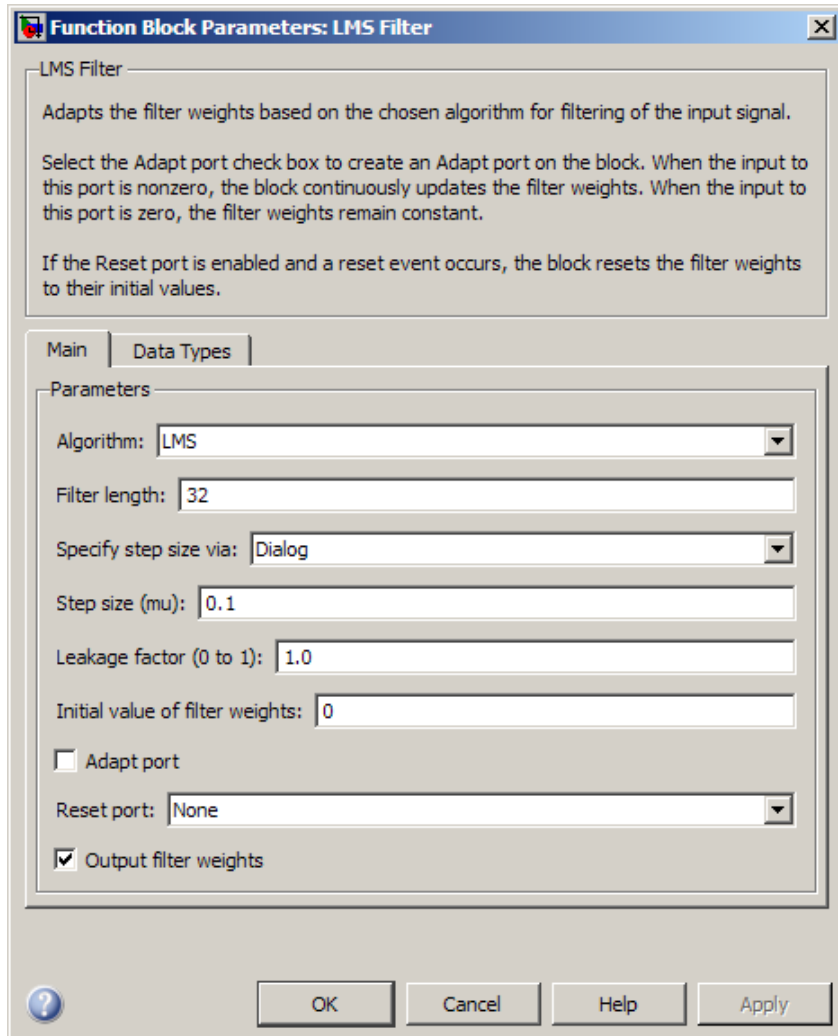
Algorithm

Choose the algorithm used to calculate the filter weights.

Filter length

Enter the length of the FIR filter weights vector.

Specify step size via

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

Step size (mu)

Enter the step size μ. Tunable.

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \leq 1$. Tunable.

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Adapt port

Select this check box to enable the Adapt input port.

Reset port

Select this check box to enable the Reset input port.

Output filter weights

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

The **Data Types** pane of the LMS Filter block dialog appears as follows.

LMS Filter

Function Block Parameters: LMS Filter

─LMS Filter───

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

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

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

| Main | Data Types |

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.

─Fixed-point operational parameters──────────────────────

Rounding mode: [Floor ▼]   Overflow mode: [Wrap ▼]

─Fixed-point data types──────────────────────────────────

　　　　　　　　　　Data Type

Parameters   [Same word length as first input ▼]

Weights   [Same as first input ▼]

Products
quotient   [Same as first input ▼]

Accumulators   [Same as first input ▼]

☐ Lock data type settings against changes by the fixed-point tools

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

Rounding mode
Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Parameters

This parameter is visible if, for the **Specify step size via** parameter, you choose `Dialog`. Choose how you specify the word length and the fraction length of the leakage factor and step size:

- When you select `Same word length as first input`, the word length of the leakage factor and step size match that of the first input to the block. In this mode, the fraction length of the leakage factor and step size is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Specify word length`, you can enter the word length of the leakage factor and step size, in bits. In this mode, the fraction length of the leakage factor and step size is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Binary point scaling`, you can enter the word length and the fraction length of the leakage factor and step size, in bits. The leakage factor and the step size must have the same word length, but the fraction lengths can differ.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the leakage factor and step size. The leakage factor and the step size must have the same word length, but the slopes can differ. This block requires a power-of-two slope and a bias of zero.

If, for the **Specify step size via** parameter, you choose `Input port`, the word length of the leakage factor is the same as the word length of the step size input at the Step size port. The fraction length of the leakage factor is automatically set to the best precision possible based on the word length of the leakage factor.

Weights

Choose how you specify the word length and fraction length of the filter weights of the block:

- When you select `Same as first input`, the word length and fraction length of the filter weights match those of the first input to the block.

- When you select `Binary point scaling`, you can enter the word length and the fraction length of the filter weights, in bits.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the filter weights. This block requires a power-of-two slope and a bias of zero.

Products & quotient

Choose how you specify the word length and fraction length of

$\mathbf{u'u}$, $\mathbf{W'u}$, $\mu \cdot e$, $Q \cdot \mathbf{u}$, and the quotient, Q. Here, $\mathbf{u}$ is the input vector, $\mathbf{W}$ is the vector of filter weights, μ is the step size, e is the

error, and Q is the quotient, which is defined as $Q = \dfrac{\mu \cdot e}{\mathbf{u'u}}$

- When you select `Same as first input`, the word length and fraction length of these quantities match those of the first input to the block.

- When you select `Binary point scaling`, you can enter the word length and the fraction length of these quantities, in bits. The word length of the quantities must be the same, but the fraction lengths can differ.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of these quantities. The word length of the quantities must be the same, but the slopes can differ. This block requires a power-of-two slope and a bias of zero.

Accumulators

Use this parameter to specify how you would like to designate the word and fraction lengths of the accumulators for the **u'u** and **W'u** operations.

Note This parameter is *not* used to designate the word and fraction lengths of the accumulator for the $Q \cdot \mathbf{u}$ operation. The accumulator data type for this quantity is automatically set to be the same as the product data type. The minimum, maximum, and overflow information for this accumulator is logged as part of the product information. Autoscaling treats this product and accumulator as one data type.

See "Fixed-Point Data Types" on page 2-899 and "Multiplication Data Types" for illustrations depicting the use of the accumulator data type in this block:

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

- When you select `Binary point scaling`, you can enter the word length and the fraction length of the accumulators, in bits. The word length of both the accumulators must be the same, but the fraction lengths can differ.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the accumulators. The word length of both the accumulators must be the same, but the slopes can differ. This block requires a power-of-two slope and a bias of zero.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

LMS Filter

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

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed point
Desired	• Must be the same as Input for floating-point signals • Must be any signed fixed-point data type when Input is fixed point
Step-size	• Must be the same as Input for floating-point signals • Must be any signed fixed-point data type when Input is fixed point
Adapt	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers
Reset	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers
Output	• Must be the same as Input for floating-point signals • Must be the same as Desired for fixed-point signals

Port	Supported Data Types
Error	• Must be the same as Input for floating-point signals • Must be the same as Desired for fixed-point signals
Wts	• Must be the same as Input for floating-point signals • Obeys the **Weights** parameter for fixed-point signals

See Also

Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
RLS Filter	DSP System Toolbox
Block LMS Filter	DSP System Toolbox
Fast Block LMS Filter	DSP System Toolbox

See "Adaptive Filters in Simulink" for related information.

Lowpass Filter

Purpose   Design lowpass filter

Library   Filtering / Filter Designs

dspfdesign

Description   This block brings the filter design capabilities of the `filterbuilder` function to the Simulink environment.

Dialog Box

View filter response

Lowpass Filter

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Impulse response

> Select either FIR or IIR from the drop-down list. FIR is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

> ---
> **Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.
> ---

Order mode

> Select Minimum (the default) or Specify. Selecting Specify enables the **Order** option so you can enter the filter order. When you set the **Impulse response** to IIR, you can specify different numerator and denominator orders. To specify a different denominator order, you must select the **Denominator order** check box.

Order

> Enter the filter order. This option is enabled only if you set the **Order mode** to Specify.

Denominator order

Select this check box to specify a different denominator order. This option is enabled only if you set the **Impulse response** to IIR and the **Order mode** to Specify.

Filter type

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to the one shown in the following figure.

Lowpass Filter

In the figure, regions between specification values such as F_{pass} and F_{stop} represent transition regions where the filter response is not constrained.

Frequency constraints

When **Order mode** is Specify, select the filter features that the block uses to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- Stopband edge and passband edge — Define the filter by specifying the frequencies for the edges for the stop- and passbands.

- Passband edge — Define the filter by specifying the edge of the passband.

- Stopband edge — Define the filter by specifying the edge of the stopband.

- Stopband edge and 3 dB point — For IIR filters, define the filter by specifying the frequency of the 3 dB point in the filter response and the edge of the stopband.

- 3 dB point — Define the filter response by specifying the location of the 3 dB point. The 3 dB point is the frequency for the point three decibels below the passband value.

- `3 dB point and passband edge` — For IIR filters, define the filter by specifying the frequency of the 3 dB point in the filter response and the edge of the passband.

- `6 dB point` — For FIR filters, define the filter response by specifying the location of the 6 dB point. The 6 dB point is the frequency for the point six decibels below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

F3dB

When **Frequency constraints** is `3 dB point`, `Stopband edge and 3 dB point`, or `3 dB point and passband edge`, specify the frequency of the 3 dB point. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Lowpass Filter

F6dB

When **Frequency constraints** is `6 dB point`, specify the frequency of the 6 dB point. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude constraints

This option is only available when you specify the order of your filter design. Depending on the setting of the **Frequency constraints** parameter, some combination of the following options will be available for the **Magnitude constraints** parameter: `Unconstrained`, `Passband ripple`, `Passband ripple and stopband attenuation` or `Stopband attenuation`.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.

- `dB` — Specify the magnitude in decibels (default)

- `Squared` — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is `Equiripple`.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Specify the phase constraint of the filter as `Linear`, `Maximum`, or `Minimum`.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select `Any`, `Even`, or `Odd` from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to Linear, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.

- When you set **Stopband shape** to 1/f, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. The block applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed    see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.

- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option,

you must set the **Input processing** parameter to Elements as channels (sample based).

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

LPC to LSF/LSP Conversion

Purpose	Convert linear prediction coefficients to line spectral pairs or line spectral frequencies
Library	Estimation / Linear Prediction
	`dsplp`

Description

The LPC to LSF/LSP Conversion block takes a vector or matrix of linear prediction coefficients (LPCs) and converts it to a vector or matrix of line spectral pairs (LSPs) or line spectral frequencies (LSFs). When converting LPCs to LSFs, the block outputs match those of the `poly2lsf` function.

The block input must be a sample-based row vector, which is treated as a single channel, or a matrix, which is treated as a single channel per column.

The input LPCs for each channel, 1, a_1, a_2, ..., a_m, must be the denominator of the transfer function of a stable all-pole filter with the form given in the first equation of "Requirements for Valid Outputs" on page 2-924. A length-M+1 input channel yields a length-M output channel. Inputs can be sample based or frame based, but outputs are always sample based.

See other sections of this reference page to learn about how to ensure that you get valid outputs, how to detect invalid outputs, how the block computes the LSF/LSP values, and more.

Requirements for Valid Outputs

To get valid outputs, your inputs and the **Root finding coarse grid points** parameter value must meet these requirements:

- The input LPCs for each channel, 1, a_1, a_2, ..., a_m, must come from the denominator of the following transfer function, $H(z)$, of a stable all-pole filter (all roots of $H(z)$ must be inside the unit circle). Note that the first term in $H(z)$'s denominator must be 1. When the input LPCs do not come from a transfer function of the following form, the block outputs are invalid.

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \; ... \; + a_m z^{-m}}$$

- The **Root finding coarse grid points** parameter value must be large enough so that the block can find all the LSP or LSF values. (The output LSFs and LSPs are roots of polynomials related to the input LPC polynomial; the block looks for these roots to produce the output. For details, see "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on page 2-932.) When you do not set **Root finding coarse grid points** to a high enough value relative to the number of LPCs, the block might not find all the LSPs or LSFs and yield invalid outputs as described in "Root Finding Method Limitations: Failure to Find Roots" on page 2-936.

To learn about recognizing invalid inputs and outputs and parameters for dealing with them, see "Handling and Recognizing Invalid Inputs and Outputs" on page 2-927.

Setting Outputs to LSFs or LSPs

Set the **Output** parameter to one of the following settings to determine whether the block outputs LSFs or LSPs:

- LSF in radians (0 pi) — Block outputs the LSF values between 0 and π radians in increasing order. The block does not output the guaranteed LSF values, 0 and π.

- LSF normalized in range (0 0.5) — Block outputs normalized LSF values in increasing order, computed by dividing the LSF values between 0 and π radians by 2π. The block does not output the guaranteed normalized LSF values, 0 and 0.5.

- LSP in range (-1 1) — Block outputs LSP values in decreasing order, equal to the cosine of the LSF values between 0 and π radians. The block does not output the guaranteed LSP values, -1 and 1.

LPC to LSF/LSP Conversion

Adjusting Output Computation Time and Accuracy with Root Finding Parameters

The values n and k determine the block's output computation time and accuracy, where

- n is the value of the **Root finding coarse grid points** parameter (choose this value with care; see the note below).

- k is the value of the **Root finding bisection refinement** parameter.

- Decreasing the values of n and k decreases the output computation time, but also decreases output accuracy:

  - The upper bound of block's computation time is proportional to $k \cdot (n-1)$.

  - Each LSP output is within $1/(n \cdot 2^k)$ of the actual LSP value.

  - Each LSF output is within ΔLSF of the actual LSF value, LSF_{act}, where

    $$\Delta LSF = \left| a\cos\left(LSF_{act}\right) - a\cos\left(LSF_{act} + 1/\left(n \cdot 2^k\right)\right) \right|$$

Note When the value of the **Root finding coarse grid points** parameter is too small relative to the number of LPCs, the block might output invalid data as described in "Requirements for Valid Outputs" on page 2-924. Also see "Handling and Recognizing Invalid Inputs and Outputs" on page 2-927.

Notable Input and Output Properties

- To get valid outputs, your input LPCs and the value of the **Root finding coarse grid points** parameter must meet the requirements described in "Requirements for Valid Outputs" on page 2-924.

- Length-L+1 input channel yields length-L output channel

- Output is always sample based

- **Output** parameter determines the output type (see "Setting Outputs to LSFs or LSPs" on page 2-925):

  - LSFs — frequencies, w_k, where $0 < w_k < \pi$ and $w_k < w_{k+1}$

  - Normalized LSFs — $w_k / 2\pi$

  - LSPs — $\cos(w_k)$

Handling and Recognizing Invalid Inputs and Outputs

The block outputs invalid data when your input LPCs and the value of the **Root finding coarse grid points** parameter do not meet the requirements described in "Requirements for Valid Outputs" on page 2-924. The following topics describe what invalid outputs look like, and how to set the block parameters provided for handling invalid inputs and outputs:

- "What Invalid Outputs Look Like" on page 2-927

- "Parameters for Handling Invalid Inputs and Outputs" on page 2-928

What Invalid Outputs Look Like

The channels of an invalid output have the same dimensions, sizes, and frame statues as the channels of a valid output. However, invalid output channels do not contain all the LSP or LSF values. Instead, they contain none or some of the LSP and LSF values and the rest of the output is filled with place holder values (-1, 0.5, or π) depending on the **Output** parameter setting).

In short, all invalid outputs in a channel end in one of the place holder values (-1, 0.5, or π) as illustrated in the following table. To learn how to use the block's parameters for handling invalid inputs and outputs, see the next section.

Output Parameter Setting	Place Holder	Sample Invalid Outputs
`LSF in radians (0 pi)`	π	$\begin{bmatrix} w_1 & w_2 & w_3 & \pi & \pi & \pi & \pi & \pi \end{bmatrix}$
`LSF normalized in range (0 0.5)`	0.5	$\begin{bmatrix} w_1 \\ w_2 \\ 0.5 \end{bmatrix}$
`LSP in range (-1 1)`	-1	$\begin{bmatrix} \cos(w_{13}) \\ \cos(w_{23}) \\ -1 \\ -1 \\ -1 \end{bmatrix}$

Parameters for Handling Invalid Inputs and Outputs

You must set how the block handles invalid inputs and outputs by setting these parameters:

- **Show output validity status (1=valid, 0=invalid)** — Set this parameter to activate a second output port that outputs a vector with one Boolean element per channel; 1 when the output of the corresponding channel is valid, and 0 when the output is invalid. The LSF and LSP outputs are invalid when the block fails to find all the LSF or LSP values or when the input LPCs are unstable (for details, see "Requirements for Valid Outputs" on page 2-924). See the previous section to learn how to recognize invalid outputs.

- **If current output is invalid, overwrite with previous output** — Select this check box to cause the block to overwrite invalid outputs with the previous output. When you set this parameter you also need to consider these parameters:

- **When first output is invalid, overwrite with user-defined values** — When the first input is unstable, you can overwrite the invalid first output with either

    · The default values, by clearing this check box

    · Values you specify, by selecting this check box

    The default initial overwrite values are the LSF or LSP representations of an all-pass filter. The vector that is used to overwrite invalid output is stored as an internal state.

- **User-defined LSP/LSF values for overwriting invalid first output** — Specify a vector of values for overwriting an invalid first output if you selected the **When first output is invalid, overwrite with user-defined values** parameter. For multichannel inputs, provide a matrix with the same number of channels as the input, or one vector that will be applied to every channel. The vector or matrix of LSP/LSF values you specify should have the same dimension, size, and frame status as the other outputs.

- **If first input value is not 1** — The block output in any channel is invalid when the first coefficient in an LPC vector is not 1; this parameter determines what the block does when given such inputs:

    - `Ignore` — Proceed with computations as if the first coefficient is 1.

    - `Normalize` — Divide the input LPCs by the value of the first coefficient before computing the output.

    - `Normalize and warn` — In addition to `Normalize`, display a warning message at the MATLAB command line.

    - `Error` — Stop the simulation and display an error message at the MATLAB command line.

LPC to LSF/LSP Conversion
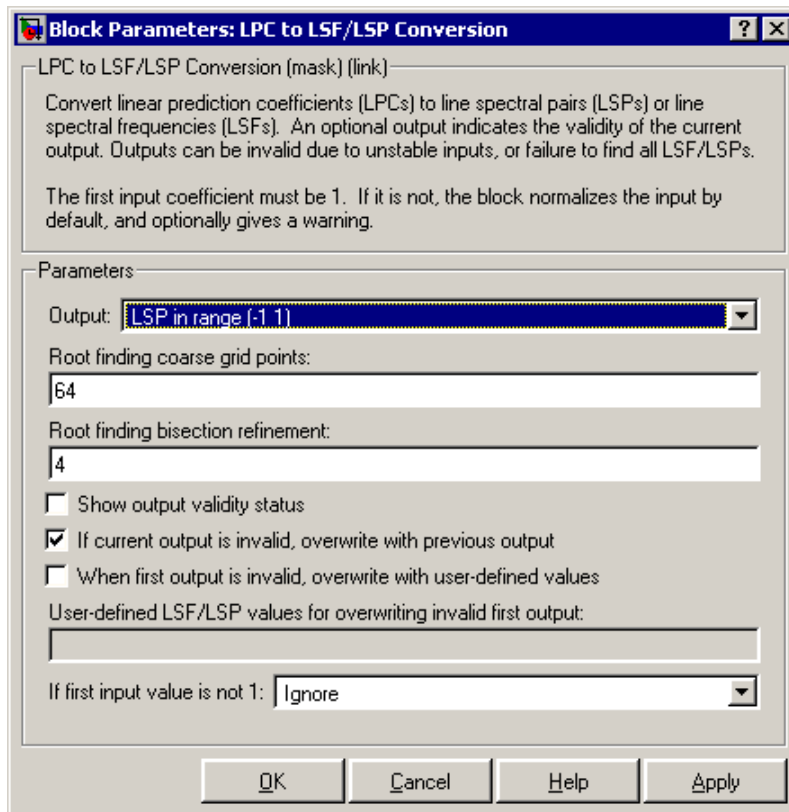
Dialog Box

Output

Specifies whether to convert the input linear prediction polynomial coefficients (LPCs) to `LSP in range (-1 1)`, `LSF in radians (0 pi)`, or `LSF normalized in range (0 0.5)`. See "Setting Outputs to LSFs or LSPs" on page 2-925 for descriptions of the three settings.

Root finding coarse grid points

The value n, where the block divides the interval (-1, 1) into n subintervals of equal length, and looks for roots (LSP values) in each subinterval. You must pick n large enough or the block

output might be invalid as described in "Requirements for Valid Outputs" on page 2-924. To learn how the block uses this parameter to compute the output, see "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on page 2-932. Also see "Adjusting Output Computation Time and Accuracy with Root Finding Parameters" on page 2-926. Tunable.

Root finding bisection refinement

The value k, where each LSP output is within $1/(n \cdot 2^k)$ of the actual LSP value, where n is the value of the **Root finding coarse grid points** parameter. To learn how the block uses this parameter to compute the output, see "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on page 2-932. Also see "Adjusting Output Computation Time and Accuracy with Root Finding Parameters" on page 2-926. Tunable.

Show output validity status

Set this parameter to activate a second output port that outputs a vector with one Boolean element per channel; 1 when the output of the corresponding channel is valid, and 0 when the output is invalid. The LSF and LSP outputs are invalid when the block fails to find all the LSF or LSP values or when the input LPCs are unstable (for details, see "Requirements for Valid Outputs" on page 2-924).

If current output is invalid, overwrite with previous output

Selecting this check box causes the block to overwrite invalid outputs with the previous output. Setting this parameter activates other parameters for taking care of initial overwrite values (when the very first output of the block is invalid). For more information, see "Parameters for Handling Invalid Inputs and Outputs" on page 2-928.

When first output is invalid, overwrite with user-defined values

When the first input is unstable, you can overwrite the invalid first output with either

• The default values, by clearing this check box

- Values you specify, by selecting this check box
The default initial overwrite values are the LSF or LSP
representations of an all-pass filter. The vector that is used to
overwrite invalid output is stored as an internal state. For more
information, see "Parameters for Handling Invalid Inputs and
Outputs" on page 2-928.

**User-defined LSP/LSF values for overwriting invalid first
output**

Specify a vector of values for overwriting an invalid first output
if you selected the **When first output is invalid, overwrite
with user-defined values** parameter. For multichannel inputs,
provide a matrix with the same number of channels as the input,
or one vector that will be applied to every channel. The vector
or matrix of LSP/LSF values you specify should have the same
dimension, size, and frame status as the other outputs.

If first input value is not 1

Determines what the block does when the first coefficient of an
input is not 1. The block can either proceed with computations
as when the first coefficient is 1 (`Ignore`); divide the input
LPCs by the value of the first coefficient before computing the
output (`Normalize`); in addition to `Normalize`, display a warning
message at the MATLAB command line (`Normalize and warn`);
stop the simulation and display an error message at the MATLAB
command line (`Error`). For more information, see "Parameters for
Handling Invalid Inputs and Outputs" on page 2-928.

Theory

**LSF and LSP Computation Method: Chebyshev Polynomial
Method for Root Finding**

Note To learn the principles on which the block's LSP and LSF
computation method is based, see the reference listed in "References"
on page 2-938.

To compute LSP outputs for each channel, the block relies on the fact that LSP values are the roots of two particular polynomials related to the input LPC polynomial; the block finds these roots using the Chebyshev polynomial root finding method, described next. To compute LSF outputs, the block computes the arc cosine of the LSPs, outputting values ranging from 0 to π radians.

Root Finding Method

LSPs, which are the roots of two particular polynomials, always lie in the range (-1, 1). (The guaranteed roots at 1 and -1 are factored out.) The block finds the LSPs by looking for a sign change of the two polynomials' values between points in the range (-1, 1). The block searches a maximum of $k(n-1)$ points, where

- n is the value of the **Root finding coarse grid points** parameter.

- k is the value of the **Root finding bisection refinement** parameter.

The block's method for choosing which points to check consists of the following two steps:

1 Coarse Root Finding —- The block divides the interval [-1, 1] into n intervals, each of length $2/n$, and checks the signs of both polynomials' values at the endpoints of the intervals. The block starts checking signs at 1, and continues checking signs at $1 - 4/n$, $1 - 6/n$, and so on at steps of length $2/n$, outputting any point if it is a root. The block stops searching in these situations:

a The block finds a sign change of a polynomial's values between two adjacent points. An interval containing a sign change is guaranteed to contain a root, so the block further searches the interval as described in Step 2, Root Finding Refinement.

b The block finds and outputs all M roots (given a length-M+1 LPC input).

c The block fails to find all M roots and yields invalid outputs as described in "Handling and Recognizing Invalid Inputs and Outputs" on page 2-927.

2 Root Finding Refinement — When the block finds a sign change in an interval, [*a*, *b*], it searches for the root guaranteed to lie in the interval by following these steps:

    a Check if Midpoint Is a Root — The block checks the sign of the midpoint of the interval [*a*, *b*]. The block outputs the midpoint if it is a root, and continues Step 1, Coarse Root Finding, at the next point, $a - 2/n$. Otherwise, the block selects the half-interval with endpoints of opposite sign (either [*a*, (*a* + *b*)/2] or [(*a* + *b*)/2, *b*]) and executes Step 2b, Stop or Continue Root Finding Refinement.

    b Stop or Continue Root Finding Refinement — When the block has repeated Step 2a *k* times (*k* is the value of the **Root finding bisection refinement** parameter), the block linearly interpolates the root by using the half-interval's endpoints, outputs the result as an LSP value, and returns to Step 1, Coarse Root Finding. Otherwise, the block repeats Step 2a using the half-interval.

Coarse Root Finding: LSPs are roots of two particular polynomials related to the input LPCs. Check signs of the two polynomials at evenly-spaced points to find all intervals containing a sign change. Output any roots (LSPs) found.

Root finding coarse grid points = 5
Divide [-1, 1] into five intervals of equal length and check signs of the polynomials' values at the endpoints of the intervals: 1, 0.6, 0.2, -0.2, -0.6, -1.

Root

Root

-1   -0.6   -0.2   0.2   0.6   1

Sign change
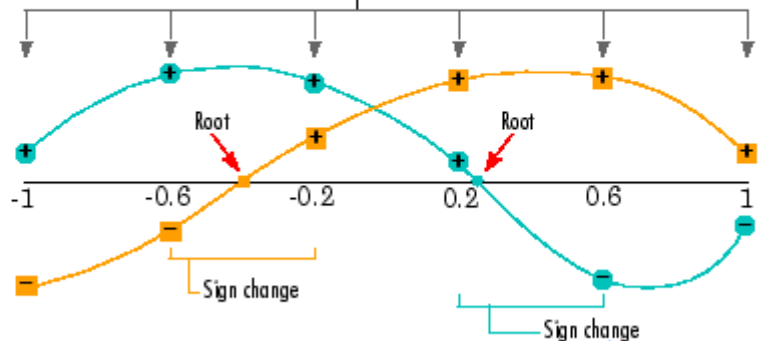
Sign change

Root Finding Refinement: Whenever Coarse Root Finding identifies an interval containing a sign change, repeatedly bisect the interval to better approximate the root (LSP value).

Bisect

Bisection 1: Check the sign of the polynomial at the midpoint of the interval and select the half-interval with endpoints of opposite sign: [0.2, 0.4]

Root

0.2   0.6

Bisection 2: Similar to Bisection 1

Root finding bisection refinement = 3
Bisect all sign change intervals found in the Coarse Root Finding up to three times to find the root. When the root is not found in the last bisection, linearly interpolate the root.

Root

0.2   0.4

Bisection 3: The last bisection. Since the midpoint of this interval is not the root, linearly interpolate the root and output the result as an LSP value.

Root (actual LSP value)

0.2   0.3

Linearly interpolated LSP approximation
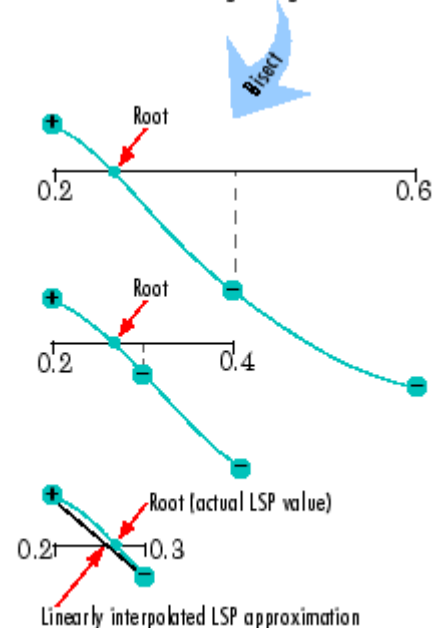
Coarse Root Finding and Root Finding Refinement

Root Finding Method Limitations: Failure to Find Roots

The block root finding method described above can fail, causing the block to produce invalid outputs (for details on invalid outputs, see "Handling and Recognizing Invalid Inputs and Outputs" on page 2-927).

In particular, the block can fail to find some roots if the value of the **Root finding coarse grid points** parameter, n, is too small. If the polynomials oscillate quickly and have roots that are very close together, the root finding might be too coarse to identify roots that are very close to each other, as illustrated in Fixing a Failed Root Finding on page 2-937.

For higher-order input LPC polynomials, you should increase the **Root finding coarse grid points** value to ensure the block finds all the roots and produces valid outputs.

Root Finding Fails: The root search divides the interval [-1, 1] into four intervals, but all three roots are in a single interval. The block can only find one root per interval, so two of the roots are never found.

Root finding coarse grid points = 4
Divide [-1, 1] into four intervals of equal length and check signs of the polynomials' values at the endpoints of the intervals: 1, 0.5, 0, -0.5, -1.

There are three roots
in a single interval

Bisect 1

Missed roots

Bisect 2

Only root found
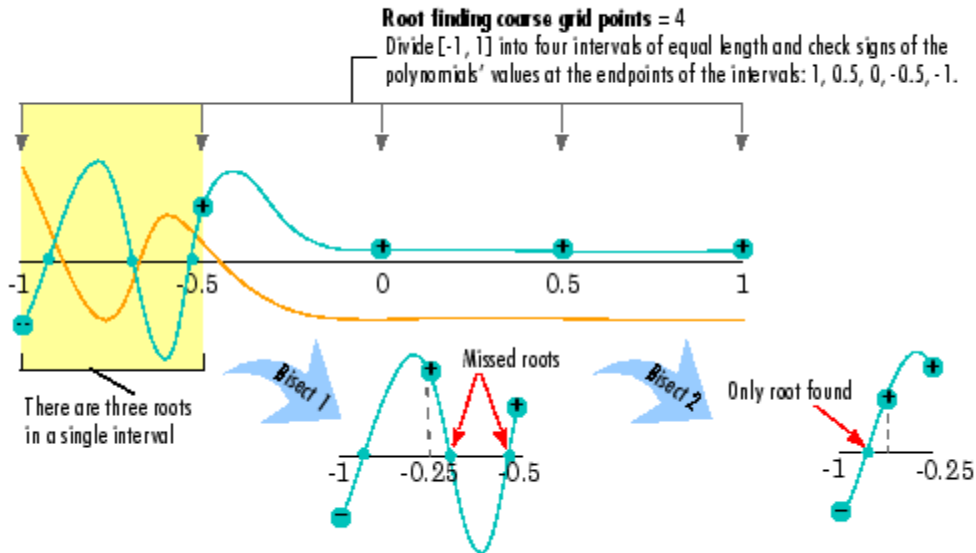
Fix Root Finding so it Succeeds: Increasing the value of the **Root finding coarse grid points** parameter to 15 ensures that each root is in its own interval, so all roots are found.

Root finding coarse grid points = 15

Each root is in its own interval.
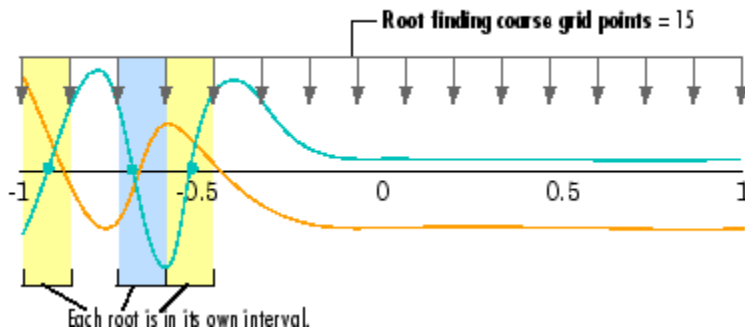
Fixing a Failed Root Finding

LPC to LSF/LSP Conversion

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Boolean — Supported only by the optional output port that appears when you set the parameter, **Show output validity status (1=valid, 0=invalid)**

References   Kabal, P. and Ramachandran, R. "The Computation of Line Spectral Frequencies Using Chebyshev Polynomials."*IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34 No. 6, December 1986. pp. 1419-1426.

See Also

LSF/LSP to LPC Conversion	DSP System Toolbox
LPC to/from RC	DSP System Toolbox
LPC/RC to Autocorrelation	DSP System Toolbox
poly2lsf	Signal Processing Toolbox

Purpose   Convert line spectral frequencies or line spectral pairs to linear prediction coefficients

Library   Estimation / Linear Prediction

dsplp

Description The LSF/LSP to LPC Conversion block takes a vector or matrix of line spectral pairs (LSPs) or line spectral frequencies (LSFs) and converts it to a vector or matrix of linear prediction polynomial coefficients (LPCs). When converting LSFs to LPCs, the block outputs match those of the lsf2poly function.

The block input must be a sample-based row vector, which is treated as a single channel, or a matrix, which is treated as a single channel per column. Each input channel must be in the same format, which you specify in the **Input** parameter:

- LSF in range (0 pi) — Vector of LSF values between 0 and π radians in increasing order. Do not include the guaranteed LSF values, 0 and π.

- LSF normalized in range (0 0.5) — Vector of normalized LSF values in increasing order, (compute by dividing the LSF values between 0 and π radians by 2π). Do not include the guaranteed normalized LSF values, 0 and 0.5.

- LSP in range (-1 1) — Vector of LSP values in decreasing order, equal to the cosine of the LSF values between 0 and π radians. Do not include the guaranteed LSP values, -1 and 1.

LSF/LSP to LPC Conversion
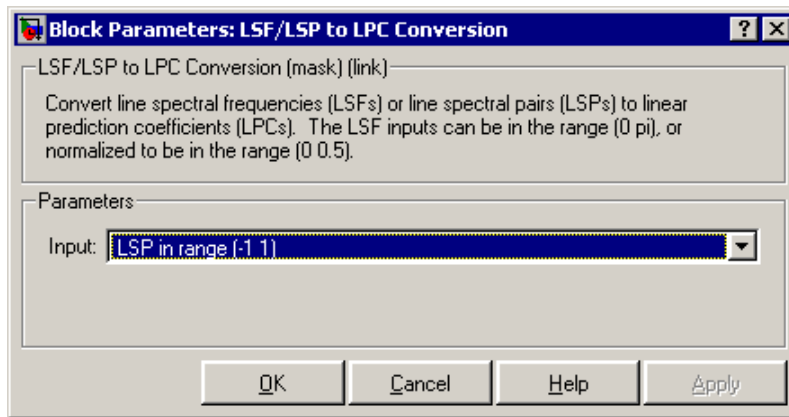
**Dialog
Box**

Input

Specifies whether to convert LSP in range (-1 1), LSF in
range (0 pi), or LSF normalized in range (0 0.5) to linear
prediction coefficients (LPCs).

**Supported
Data
Types**

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

References

Kabal, P. and Ramachandran, R. "The Computation of Line Spectral
Frequencies Using Chebyshev Polynomials." *IEEE Transactions
on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34 No. 6,
December 1986. pp. 1419-1426.

See Also

LPC to LSF/LSP Conversion	DSP System Toolbox
LPC to/from RC	DSP System Toolbox
LPC/RC to Autocorrelation	DSP System Toolbox
lsf2poly	Signal Processing Toolbox

Purpose   Convert linear prediction coefficients to cepstral coefficients or cepstral coefficients to linear prediction coefficients

Library   Estimation / Linear Prediction

dsplp
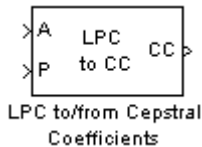
Description   The LPC to/from Cepstral Coefficients block either converts linear prediction coefficients (LPCs) to cepstral coefficients (CCs) or cepstral coefficients to linear prediction coefficients. Set the **Type of conversion** parameter to LPCs to cepstral coefficients or Cepstral coefficients to LPCs to select the domain into which you want to convert your coefficients. The LPC port corresponds to LPCs, and the CC port corresponds to the CCs. For more information, see "Algorithm" on page 2-942.

The block input must be a sample-based row vector, which is treated as a single channel, or a matrix, which is treated as a single channel per column.

Consider a signal $x(n)$ as the input to an FIR analysis filter represented by LPCs. The output of this analysis filter, $e(n)$, is known as the prediction error signal. The power of this error signal is denoted by P, the prediction error power.

When you select LPCs to cepstral coefficients from the **Type of conversion** list, you can specify the prediction error power in two ways. From the **Specify P** list, choose via input port to input the prediction error power using input port P. The input to the port must be a vector with length equal to the number of input channels. Select assume P equals 1 to set the prediction error power equal to 1 for all channels.

When you select LPCs to cepstral coefficients from the **Type of conversion** list, the **Output size same as input size** check box appears. When you select this check box, the length of the input vector of LPCs is equal to the output vector of CCs. When you do not select this check box, enter a positive scalar for the **Length of output cepstral coefficients** parameter.

2-941

LPC to/from Cepstral Coefficients

When you select `LPCs to cepstral coefficients` from the **Type of conversion** list, you can use the **If first input value is not 1** parameter to specify the behavior of the block when the first coefficient of the LPC vector is not 1. The following options are available:

- `Replace it with 1` —- Changes the first value of the coefficient vector to 1. The other coefficient values are unchanged.

- `Normalize` — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC vector is 1.

- `Normalize and Warn` — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.

- `Error` — Displays an error telling you that the first coefficient of the LPC vector is not 1.

When you select `Cepstral coefficients to LPCs` from the **Type of conversion** list, the **Output P** check box appears on the block. Select this check box when you want to output the prediction error power from output port P.

Algorithm   The cepstral coefficients are the coefficients of the Fourier transform representation of the logarithm magnitude spectrum. Consider a sequence, $x(n)$, having a Fourier transform $X(\omega)$. The cepstrum, $c_x(n)$, is defined by the inverse Fourier transform of $C_x(\omega)$, where $C_x(\omega) = \log_e X(\omega)$. See the Real Cepstrum block reference page for information on computing cepstrum coefficients from time-domain signals.

LPC to CC

When in this mode, this block uses a recursion technique to convert LPCs to CCs. The LPC vector is defined by $\begin{bmatrix} a_0 & a_1 & a_2 & ... & a_p \end{bmatrix}$ and the CC vector is defined by $\begin{bmatrix} c_0 & c_1 & c_2 & ... & c_p & ... & c_{n-1} \end{bmatrix}$. The recursion is defined by the following equations:

$$c_0 = \log_e P$$

$$c_m = -a_m + \frac{1}{m} \sum_{k=1}^{m-1} \Big[-(m-k) \cdot a_k \cdot c_{(m-k)} \Big], 1 \le m \le p$$

$$c_m = \sum_{k=1}^{p} \left[\frac{-(m-k)}{m} \cdot a_k \cdot c_{(m-k)} \right], p < m < n$$

CC to LPC

When in this mode, this block uses a recursion technique to convert CCs to LPCs. The CC vector is defined by $\begin{bmatrix} c_0 & c_1 & c_2 & ... & c_p & ... & c_n \end{bmatrix}$ and the LPC vector is defined by $\begin{bmatrix} a_0 & a_1 & a_2 & ... & a_p \end{bmatrix}$. The recursion is defined by the following equations

$$a_m = -c_m - \frac{1}{m} \sum_{k=1}^{m-1} \Big[(m-k) \cdot c_{(m-k)} \cdot a_k \Big]$$

$$P = \exp(C_0)$$

where $m = 1, 2, ..., p$.

LPC to/from Cepstral Coefficients
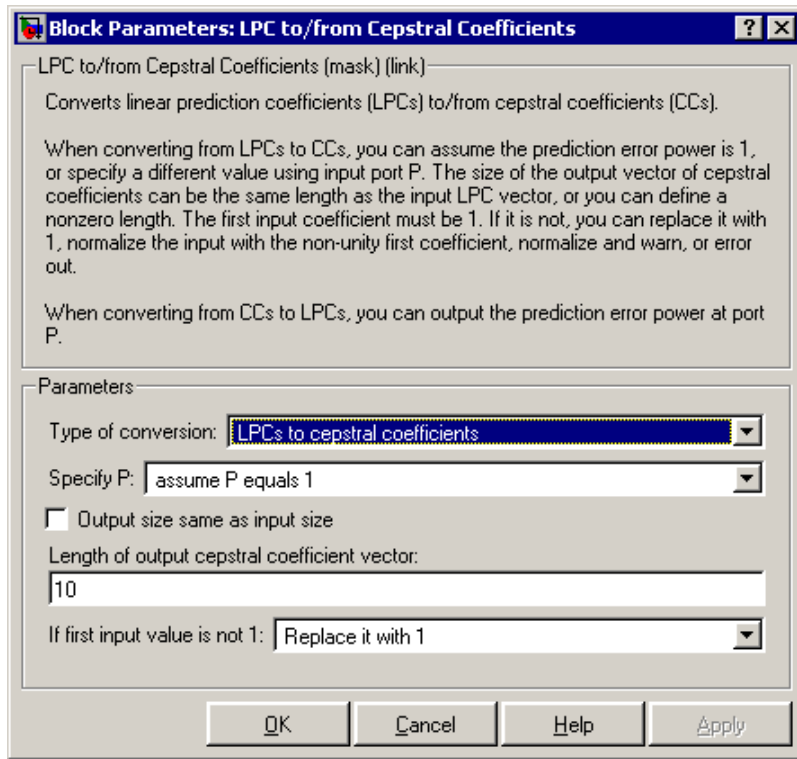
**Dialog
Box**

Type of conversion
>   Choose `LPCs to cepstral coefficients` or `Cepstral
>   coefficients to LPCs` to specify the domain into which you
>   want to convert your coefficients.

Specify P
>   Choose `via input port` to input the values of prediction error
>   power using input port P. Select `assume P equals 1` to set the
>   prediction error power equal to 1.

Output size same as input size
>   When you select this check box, the length of the input vector of
>   LPCs is equal to the output vector of CCs.

Length of output cepstral coefficients

Enter a positive scalar that is the length of each output channel of CCs.

If first input value is not 1

Select what you would like the block to do when the first coefficient of the LPC vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Normalize and Warn`, and `Error`.

Output P

Select this check box to output the prediction error power for each channel from output port P.

References   Papamichalis, Panos E. *Practical Approaches to Speech Coding*. Englewood Cliffs, NJ: Prentice Hall, 1987.

Supported Data Types

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

See Also

Levinson-Durbin	DSP System Toolbox
LPC to LSF/LSP Conversion	DSP System Toolbox
LSF/LSP to LPC Conversion	DSP System Toolbox
LPC to/from RC	DSP System Toolbox
LPC/RC to Autocorrelation	DSP System Toolbox
Real Cepstrum	DSP System Toolbox
Complex Cepstrum	DSP System Toolbox

LPC to/from RC

Purpose   Convert linear prediction coefficients to reflection coefficients or reflection coefficients to linear prediction coefficients

Library   Estimation / Linear Prediction

dsplp
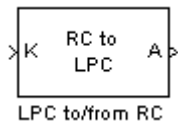
Description The LPC to/from RC block either converts linear prediction coefficients (LPCs) to reflection coefficients (RCs) or reflection coefficients to linear prediction coefficients. Set the **Type of conversion** parameter to LPC to RC or RC to LPC to select the domain into which you want to convert your coefficients. The A port corresponds to LPC coefficients, and the K port corresponds to the RC coefficients. For more information, see "Algorithm" on page 2-947.

The block input must be a sample-based row vector, which is treated as a single channel, or a matrix, which is treated as a single channel per column.

Consider a signal $x(n)$ as the input to an FIR analysis filter represented by LPC coefficients. The output of the this analysis filter, $e(n)$, is known as the prediction error signal. The power of this error signal is denoted by P. When the zero lag autocorrelation coefficient of $x(n)$ is one, the autocorrelation sequence and prediction error power are said to be normalized.

Select the **Output normalized prediction error power** check box to enable port P. The normalized prediction error power output at P is a vector with one element per input channel. Each element varies between zero and one.

Select the **Output LPC filter stability** check box to output the stability of the filter represented by the LPCs or RCs. The synthesis filter represented by the LPCs is stable when the absolute value of each of the roots of the LPC polynomial is less than one. The lattice filter represented by the RCs is stable when the absolute value of each reflection coefficient is less than 1. When the filter is stable, the block outputs a Boolean value of 1 for each input channel at the S port. When

the filter is unstable, the block outputs a Boolean value of `0` for each input channel at the S port.

If first input value is not 1 parameter specifies the behavior of the block when the first coefficient of the LPC coefficient vector in any channel is not 1. The following options are available:

- `Replace it with 1` — Changes the first value of the coefficient channel to 1. The other coefficient values are unchanged.

- `Normalize` — Divides the entire channel of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1.

- `Normalize and Warn` — Divides the entire channel of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.

- `Error` — Displays an error telling you that the first coefficient of the LPC coefficient channel is not 1.

Algorithm

LPC to RC

When in this mode, this block uses backward Levinson recursion to convert linear prediction coefficients (LPCs) to reflection coefficients (RCs). For a given Nth order LPC vector

$LPC_N = \begin{bmatrix} 1 & a_{N1} & a_{N2} & ... & a_{NN} \end{bmatrix}$, the block calculates the Nth reflection coefficient value using the formula $\gamma_N = -a_{NN}$. The block then finds the lower order LPC vectors , LPC_{N-1}, LPC_{N-2}, ..., LPC_1, using the following recursion.

for $p = N, N - 1, ..., 2,$

$$\gamma_p = a_{pp}$$
$$F = 1 - \gamma_p^2$$
$$a_{p-1,m} = \frac{a_{p,m}}{F} - \frac{\gamma_p a_{p,p-m}}{F}, \quad 1 \le m < p$$

end

Finally, $\gamma_1 = -a_{11}$. The reflection coefficient vector is $\begin{bmatrix} \gamma_1, & \gamma_2, & ..., & \gamma_N \end{bmatrix}$.

RC to LPC

When in this mode, this block uses Levinson recursion to convert reflection coefficients (RCs) to linear prediction coefficients (LPCs).

In this case, the input to the block is $RC = \begin{bmatrix} \gamma_1 & \gamma_2 & ... & \gamma_N \end{bmatrix}$. The zeroth order LPC vector term is 1. Starting with this term, the block uses recursion to calculate the higher order LPC vectors,

$LPC_2, LPC_3,...LPC_N$, until it has calculated the entire LPC matrix.

$$LPC_{matrix} = \begin{bmatrix} LPC_0 \\ LPC_1 \\ LPC_2 \\ ... \\ LPC_N \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & a_{11} & 0 & 0 & \cdots & 0 \\ 1 & a_{21} & a_{22} & 0 & \cdots & 0 \\ 1 & a_{31} & a_{32} & a_{33} & \cdots & 0 \\ ... & ... & ... & ... & ... & ... \\ 1 & a_{N1} & a_{N2} & a_{N3} & \cdots & a_{NN} \end{bmatrix}$$

This LPC matrix consists of LPC vectors of order 0 through N found by using the Levinson recursion. The following are the formulas for the recursion steps, for $p = 0, 1, ...,N - 1$.

$$a_{p+1,m} = a_{p,m} + \gamma_{p+1} a_{p,p+1-m}, \ \ 1 \le m \le p$$
$$a_{p+1,p+1} = \gamma_{p+1}$$

Dialog Box

Function Block Parameters: LPC to/from RC

LPC to/from RC (mask) (link)

Converts Linear Prediction polynomial coefficients (A) to/from Reflection coefficients (K). The normalized prediction error power of the linear prediciton filter is given in the optional P output. The stability of the prediction coefficients is given in the optional S output.

In the LPC to RC conversion, the first input coefficient must be 1. If it is not, you can optionally replace it with 1, or normalize the input with the non-unity 1st coefficient and warn, or error out.

Parameters

Type of conversion [RC to LPC]

☑ Output normalized prediction error power

☐ Output LPC filter stability

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

Type of conversion
Select LPC to RC or RC to LPC to select the domain into which you want to convert your coefficients.

Output normalized prediction error power
Select this check box to output the normalized prediction error power at port P.

Output LPC filter stability
Select this check box to output the stability of the filter. When the filter represented by the LPCs or RCs is stable, the block outputs a Boolean value of 1 for each input channel at the S port. When the filter represented by the LPCs or RCs is unstable, the block outputs a Boolean value of 0 for each input channel at the S port.

LPC to/from RC

If first input value is not 1

Select what you would like the block to do when the first coefficient of the LPC coefficient vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Normalize and Warn`, and `Error`.

References   Makhoul, J *Linear Prediction: A tutorial review.* Proc. IEEE. 63, 63, 56 (1975).

Markel, J.D. and A. H. Gray, Jr., *Linear Prediction of Speech.* New York, Springer-Verlag, 1976.

Supported Data Types

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

See Also

Levinson-Durbin	DSP System Toolbox
LPC to LSF/LSP Conversion	DSP System Toolbox
LSF/LSP to LPC Conversion	DSP System Toolbox
LPC/RC to Autocorrelation	DSP System Toolbox

Purpose   Convert linear prediction coefficients or reflection coefficients to autocorrelation coefficients

Library   Estimation / Linear Prediction

dsplp

Description The LPC/RC to Autocorrelation block either converts linear prediction coefficients (LPCs) to autocorrelation coefficients (ACs) or reflection coefficients (RCs) to autocorrelation coefficients (ACs). Set the **Type of conversion** parameter to LPC to autocorrelation or RC to autocorrelation to select the domain from which you want to convert your coefficients. The A port corresponds to LPC coefficients, and the K port corresponds to the RC coefficients.

The block input must be a sample-based row vector, which is treated as a single channel, or a matrix, which is treated as a single channel per column.

Use the **Specify P** parameter to set the value of the prediction error power. You can set this parameter to 1 by selecting Assume P=1. When you select Via input port, a P port appears on the block. You can use this port to input the value of the actual, non-unity prediction error power for each channel. The length of this vector must equal the number of channels in the input.

The **If first input value is not 1** parameter specifies the behavior of the block when the first coefficient of the LPC coefficient vector is not 1. The following options are available:

- Replace it with 1 — The block changes the first value of the coefficient vector to 1. The rest of the coefficient values are unchanged.

- Normalize — The block divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1.

- `Normalize and Warn` — The block divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.

- `Error` — The block displays an error telling you that the first coefficient of the LPC coefficient vector is not 1.
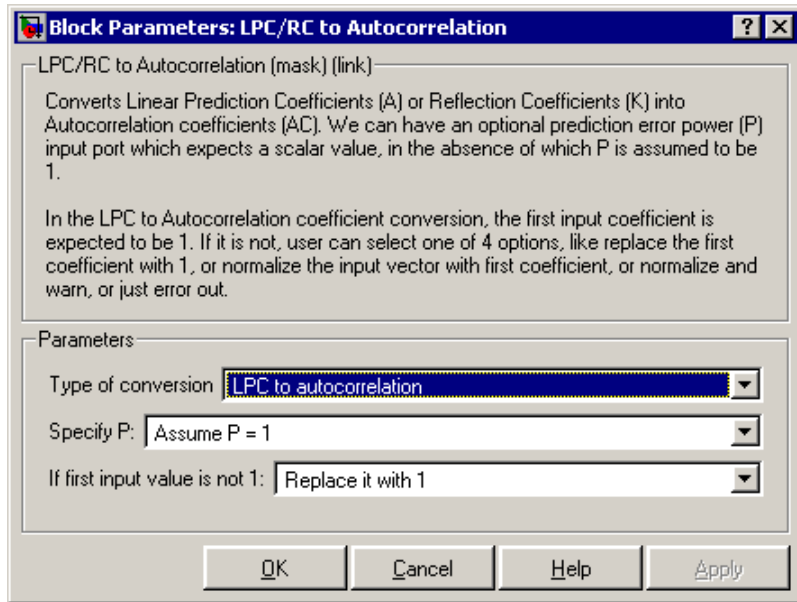
Dialog Box

Type of conversion

From the list select `LPC to autocorrelation` or `RC to autocorrelation` to specify the domain from which you want to convert your coefficients.

Specify P

From the list select `Assume P=1` or `Via input port` to specify the value of prediction error power.

If first input value is not 1

Select what you would like the block to do when the first coefficient of the LPC coefficient vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Normalize and Warn`, and `Error`.

References   Orfanidis, S.J. *Optimum Signal Processing*. New York, McGraw-Hill, 1988.

Makhoul, J. *Linear Prediction: A tutorial review*. Proc. IEEE. 63, 63, 56 (1975).

Markel, J.D. and A. H. Gray, Jr., *Linear Prediction of Speech*. New York, Springer-Verlag, 1976.

Supported Data Types

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

See Also

Levinson-Durbin	DSP System Toolbox
LPC to LSF/LSP Conversion	DSP System Toolbox
LSF/LSP to LPC Conversion	DSP System Toolbox
LPC to/from RC	DSP System Toolbox

LU Factorization

Purpose   Factor square matrix into lower and upper triangular components

Library   Math Functions / Matrices and Linear Algebra / Matrix Factorizations

dspfactors

Description
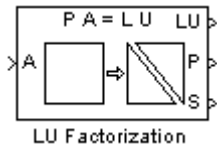
LU Factorization

The LU Factorization block factors a row-permuted version of the square input matrix A as $A_p = L*U$, where L is a unit-lower triangular matrix, U is an upper triangular matrix, and A_p contains the rows of A permuted as indicated by the permutation index vector P. The block uses the pivot matrix A_p instead of the exact input matrix A because it improves the numerical accuracy of the factorization. You can determine the singularity of the input matrix A by enabling the optional output port S. When A is singular, the block outputs a 1 at port S; when A is nonsingular, it outputs a 0.

To improve efficiency, the output of the LU Factorization block at port LU is a composite matrix containing both the lower triangle elements of L and the upper triangle elements of U. Thus, the output is in a different format than the output of the MATLAB lu function, which returns L and U as separate matrices. To convert the output from the block's LU port to separate L and U matrices, use the following code:

```
L = tril(LU,-1)+eye(size(LU));
U = triu(LU);
```

If you compare the results produced by these equations to the actual output of the MATLAB lu function, you may see slightly different values. These differences are due to rounding error, and are expected.

See the lu function reference page in the MATLAB documentation for more information about LU factorizations.

Fixed-Point Data Types

The following diagram shows the data types used within the LU Factorization block for fixed-point signals.
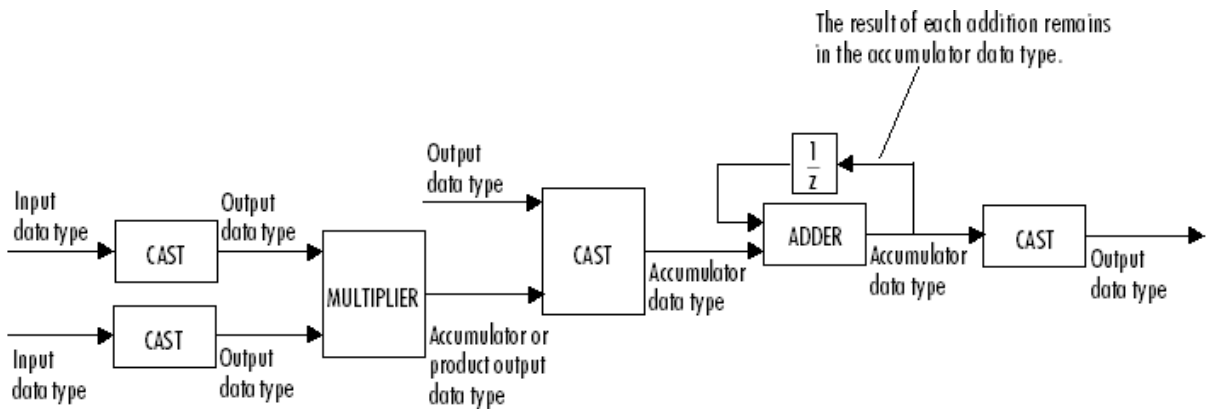
The result of each addition remains in the accumulator data type.

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

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

Examples

The row-pivoted matrix A_p and permutation index vector P computed by the block are shown below for 3-by-3 input matrix A.

$$A = \begin{bmatrix} -1 & 8 & -5 \\ 9 & -1 & 2 \\ 2 & -5 & 7 \end{bmatrix} \qquad P = \begin{pmatrix} 2 & 1 & 3 \end{pmatrix} \qquad A_P = \begin{bmatrix} 9 & -1 & 2 \\ -1 & 8 & -5 \\ 2 & -5 & 7 \end{bmatrix}$$

The LU output is a composite matrix whose lower subtriangle forms L and whose upper triangle forms U.

LU Factorization

$$\begin{bmatrix} -1 & 8 & -5 \\ 9 & -1 & 2 \\ 2 & -5 & 7 \end{bmatrix} \Rightarrow$$

$$\begin{bmatrix} 9.00 & -1.00 & 2.00 \\ -0.11 & 7.89 & -4.78 \\ 0.22 & -0.61 & 3.66 \end{bmatrix}$$

$$(2 \quad 1 \quad 3)$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.11 & 1 & 0 \\ 0.22 & -0.61 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 9.00 & -1.00 & 2.00 \\ 0 & 7.89 & -4.78 \\ 0 & 0 & 3.66 \end{bmatrix}$$
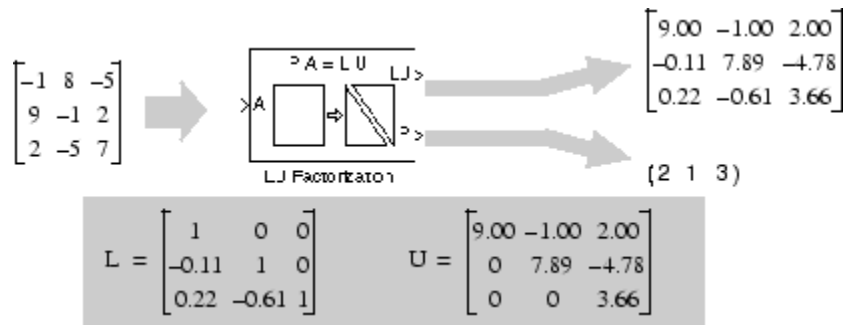
See "Factor a Matrix into Upper and Lower Submatrices Using the LU Factorization Block" in the *DSP System Toolbox User's Guide* for another example using the LU Factorization block.

**Dialog
Box**

The **Main** pane of the LU Factorization block dialog appears as follows.
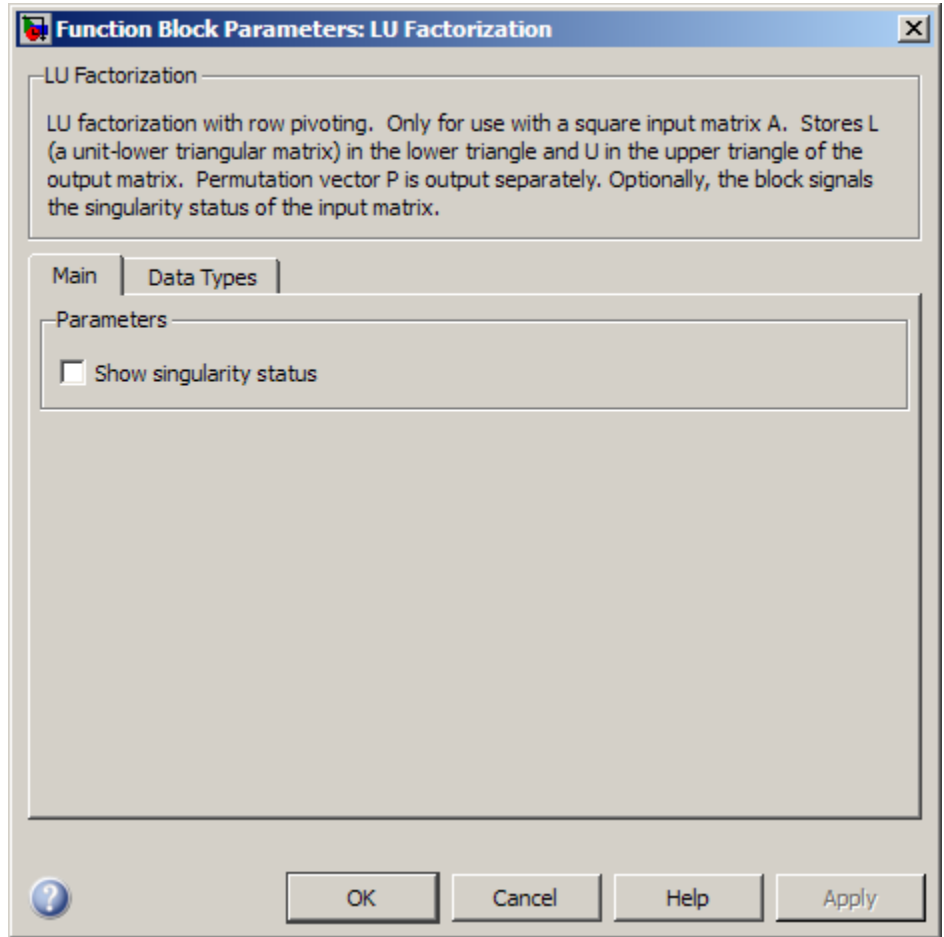
Show singularity status

> Select to output the singularity of the input at port S, which
> outputs Boolean data type values of 1 or 0. An output of 1

indicates that the current input is singular, and an output of 0 indicates the current input is nonsingular.

The **Data Types** pane of the LU Factorization block dialog appears as follows.
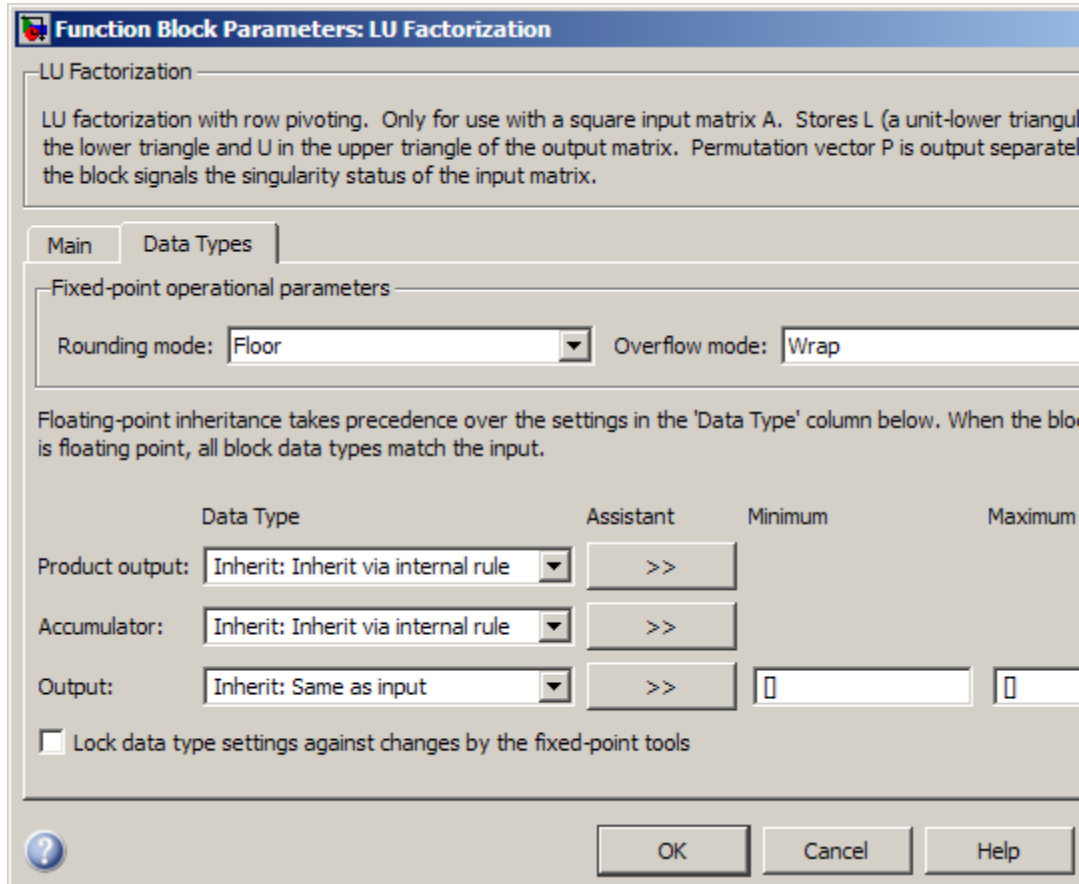
Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-954 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-954 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

LU Factorization

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-954 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

References Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
LU	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers
P	• Double-precision floating point
	• Single-precision floating point
	• 32-bit unsigned integers
S	• Boolean

See Also

Autocorrelation LPC	DSP System Toolbox
Cholesky Factorization	DSP System Toolbox
LDL Factorization	DSP System Toolbox
LU Inverse	DSP System Toolbox
LU Solver	DSP System Toolbox
Permute Matrix	DSP System Toolbox
QR Factorization	DSP System Toolbox
lu	MATLAB

See "Matrix Factorizations" for related information.

LU Inverse

Purpose   Compute inverse of square matrix using LU factorization

Library   Math Functions / Matrices and Linear Algebra / Matrix Inverses

dspinverses
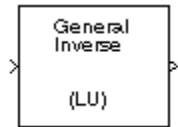
Description   The LU Inverse block computes the inverse of the square input matrix A by factoring and inverting row-pivoted variant A_p.

General
Inverse

(LU)

$$A_p^{-1} = (LU)^{-1}$$

L is a lower triangular square matrix with unity diagonal elements, and U is an upper triangular square matrix. The block outputs the inverse matrix A^{-1}.

Examples   See "Find the Inverse of a Matrix Using the LU Inverse Block" in the *DSP System Toolbox User's Guide*.
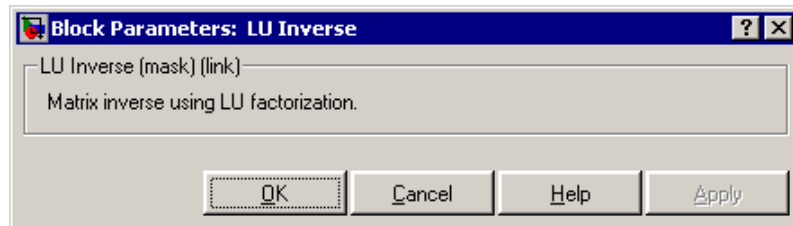
Block Parameters: LU Inverse	? X

LU Inverse (mask) (link)

Matrix inverse using LU factorization.

OK   Cancel   Help   Apply

**Dialog
Box**

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

**Supported
Data
Types**
- Double-precision floating point
- Single-precision floating point

See Also

Cholesky Inverse	DSP System Toolbox
LDL Inverse	DSP System Toolbox
LU Factorization	DSP System Toolbox
LU Solver	DSP System Toolbox
inv	MATLAB

See "Matrix Inverses" for related information.

LU Solver

Purpose

Solve *AX=B* for *X* when *A* is square matrix

Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

dspsolvers

Description
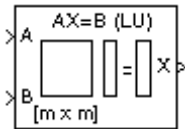
The LU Solver block solves the linear system *AX=B* by applying LU factorization to the *M*-by-*M* matrix at the A port. The input to the B port is the right side *M*-by-*N* matrix, *B*. The *M*-by-*N* matrix output *X* is the unique solution of the equations.

The block treats length-*M* unoriented vector input to the input port *B* 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.

$$LY = B_p$$
$$UX = Y$$

Examples   See "Solve AX=B Using the LU Solver Block" in the *DSP System Toolbox User's Guide*.
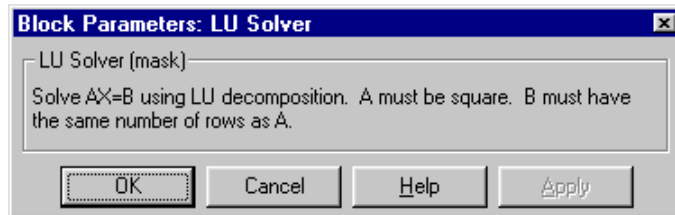
Dialog Box

Supported Data Types

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

See Also

Autocorrelation LPC	DSP System Toolbox
Cholesky Solver	DSP System Toolbox
LDL Solver	DSP System Toolbox
Levinson-Durbin	DSP System Toolbox
LU Factorization	DSP System Toolbox
LU Inverse	DSP System Toolbox
QR Solver	DSP System Toolbox

See "Linear System Solvers" for related information.

Magnitude FFT

Purpose   Compute nonparametric estimate of spectrum using periodogram method

Library   • Estimation / Power Spectrum Estimation

dspspect3

• Transforms

dspxfrm3
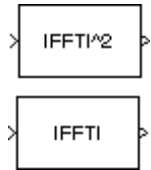
Description   The Magnitude FFT block computes a nonparametric estimate of the spectrum using the periodogram method.

When the **Output** parameter is set to Magnitude squared, the block output for an *M*-by-*N* input *u* is equivalent to

```
y = abs(fft(u,nfft)).^2      % M   nfft
```

When the **Output** parameter is set to Magnitude, the block output for an input *u* is equivalent to

```
y = abs(fft(u,nfft))         % M   nfft
```

When $M > N_{fft}$, the block wraps the input to N_{fft} before computing the FFT using one of the above equations:

```
y(:,k)=datawrap(u(:,k),nfft)     % 1   k   N
```

The block treats an *M*-by-*N* matrix input 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. 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 represents the signal's sample frequency. The block always outputs sample–based data.

The Magnitude FFT block supports real and complex floating-point inputs. The block also supports real fixed-point inputs in both

`Magnitude` and `Magnitude squared` modes, and complex fixed-point inputs in the `Magnitude squared` mode.

Fixed-Point Data Types

The following diagram shows the data types used within the Magnitude FFT subsystem block for fixed-point signals.
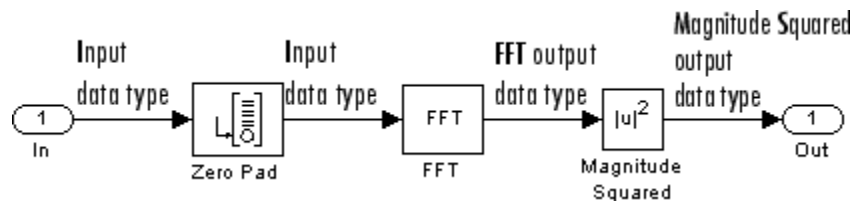
The settings for the fixed-point parameters of the FFT block in the diagram above are as follows:

- Sine table — `Same word length as input`

- Integer rounding mode — `Floor`

- Saturate on integer overflow — unchecked

- Product output — `Inherit via internal rule`

- Accumulator — `Inherit via internal rule`

- Output — `Inherit via internal rule`

The settings for the fixed-point parameters of the Magnitude Squared block in the diagram above are as follows:

- Integer rounding mode — `Floor`

- Saturate on integer overflow — checked

- Output — `Inherit via internal rule`

Examples   The `dspsacomp` example compares the periodogram method with several other spectral estimation methods.

Magnitude FFT
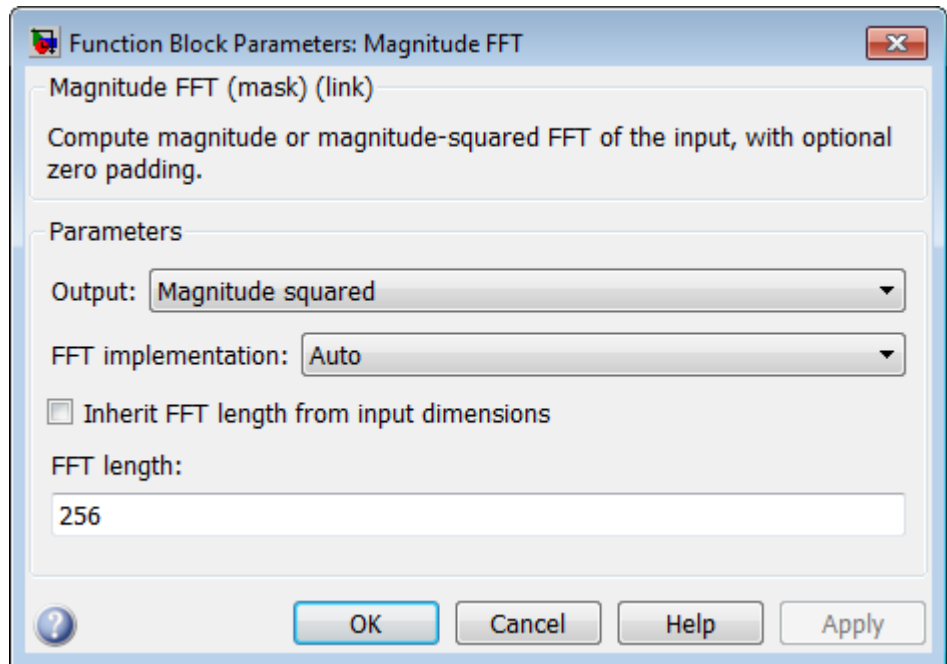
**Dialog
Box**

Output

Specify whether the block computes the magnitude FFT or
magnitude-squared FFT of the input.

FFT implementation

Set this parameter to FFTW [1], [2] to support an arbitrary length
input signal. The block restricts generated code with FFTW
implementation to MATLAB host computers.

Set this parameter to Radix-2 for bit-reversed processing, fixed
or floating-point data, or for portable C-code generation using
the Simulink Coder. The first dimension *M*, of the input matrix
must be a power of two. To work with other input sizes, use the
Pad block to pad or truncate these dimensions to powers of two,
or if possible choose the FFTW algorithm.

Set this parameter to `Auto` to let the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

Inherit FFT length from input dimensions
Select to use the input frame size as the number of data points, on which to perform the FFT. When you select this check box, this number must be a power of two. When you do not select this check box, the **FFT length** parameter specifies the number of data points.

FFT length
Enter the number of data points on which to perform the FFT, N_{fft}. When N_{fft} is larger than the input frame size, each frame is zero-padded as needed. When N_{fft} is smaller than the input frame size, each frame is wrapped as needed. This parameter is enabled when you clear the **Inherit FFT length from input dimensions** check box.

When you set the **FFT implementation** parameter to `Radix-2`, this value must be a power of two.

References Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

Orfanidis, S. J. *Introduction to Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1995.

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

References

[1] FFTW (http://www.fftw.org)

Magnitude FFT

[2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

Burg Method	DSP System Toolbox
Short-Time FFT	DSP System Toolbox
Spectrum Scope	DSP System Toolbox
Yule-Walker Method	DSP System Toolbox
pwelch	Signal Processing Toolbox

See "Spectral Analysis" for related information.

Purpose   Compute 1-norm of matrix

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

Description   The Matrix 1-Norm block computes the 1-norm, or maximum column-sum, of an *M*-by-*N* input matrix, *A*.

$$y = \|A\|_1 = \frac{\max}{1 \le j \le N} \sum_{i=1}^{M} |a_{ij}|$$

This is equivalent to

```
y = max(sum(abs(A)))        % Equivalent
MATLAB code
```


$$\|A\|_1 = max(A_1, A_2, A_3)$$

$$|a_{13}| + |a_{23}| + |a_{33}| = A_3$$

$$|a_{12}| + |a_{22}| + |a_{32}| = A_2$$

$$|a_{11}| + |a_{21}| + |a_{31}| = A_1$$
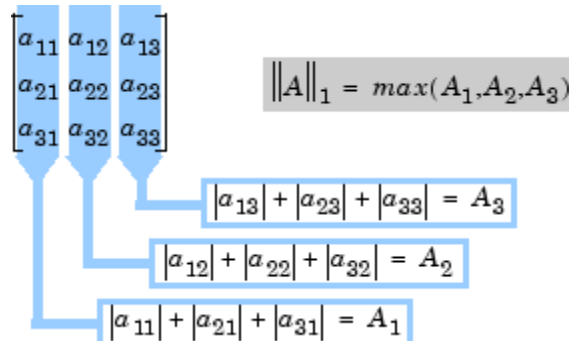
The block treats length-*M* unoriented vector input as an *M*-by-1 matrix. The output, *y*, is always a scalar.

The Matrix 1-Norm block supports real and complex floating-point inputs, and real fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Matrix 1-Norm block for fixed-point signals.

Matrix 1-Norm

The block calculations are all done in the accumulator data type until the max is performed. The result is then cast to the output data type. You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 2-972 below.

Dialog Box

There are no parameters on the **Main** pane of this dialog.

The **Data types** pane of the Matrix 1-Norm block dialog appears as follows.

Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode
> Select the rounding mode for fixed-point operations.

Overflow mode
> Select the overflow mode for fixed-point operations.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-971 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-971 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Matrix 1-Norm

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Normalization	DSP System Toolbox
Reciprocal Condition	DSP System Toolbox
norm	MATLAB

Purpose	Concatenate input signals of same data type to create contiguous output signal
Library	Math Functions / Matrices and Linear Algebra / Matrix Operations
	dspmtrx3
Description	The Matrix Concatenate block is an implementation of the Simulink Matrix Concatenate block. See Matrix Concatenate for more information.

Matrix Exponential

Purpose   Compute matrix exponential

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

Description   The Matrix Exponential block computes the matrix exponential using a scaling and squaring algorithm with a Pade approximation. The input matrix must be square.
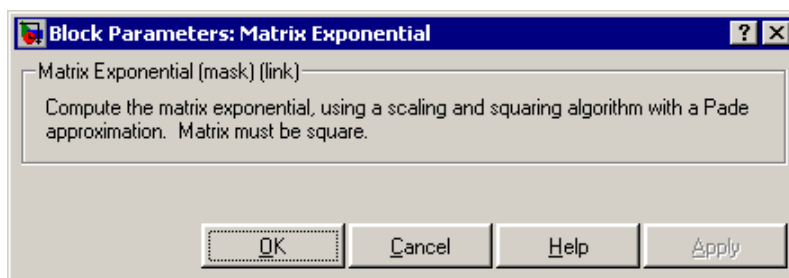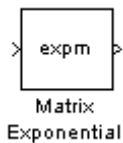
Dialog Box

Supported Data Types
- Double-precision floating point
- Single-precision floating point

See Also

Array-Vector Multiply	DSP System Toolbox
expm	MATLAB
Dot Product	Simulink

Matrix Product   DSP System Toolbox

Product   Simulink

Matrix Multiply

Purpose   Multiply or divide inputs

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

Description   The Matrix Multiply block is an implementation of the Simulink Product block. See Product for more information.

Purpose Multiply matrix elements along rows, columns, or entire input

Library Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3
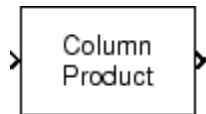
Description The Matrix Product block multiplies the elements of an *M*-by-*N* input matrix *u* along its rows, its columns, or over all its elements.

When the **Multiply over** parameter is set to Rows, the block multiplies across the elements of each row and outputs the resulting *M*-by-1 matrix. The block treats length-*N* unoriented vector input as a 1-by-*N* matrix.

$$
\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \left(\prod_{j=1}^{3} u_{1j} \right) \\ \left(\prod_{j=1}^{3} u_{2j} \right) \\ \left(\prod_{j=1}^{3} u_{3j} \right) \end{bmatrix}
$$

When the **Multiply over** parameter is set to Columns, the block multiplies down the elements of each column and outputs the resulting 1-by-*N* matrix. The block treats length-*M* unoriented vector input as an *M*-by-1 matrix.

$$
\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}
$$
$$
\Downarrow
$$
$$
\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \left[\left(\prod_{i=1}^{3} u_{i1} \right) \left(\prod_{i=1}^{3} u_{i2} \right) \left(\prod_{i=1}^{3} u_{i3} \right) \right]
$$

Matrix Product

When the **Multiply over** parameter is set to `Entire input`, the block multiplies all the elements of the input together and outputs the resulting scalar.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow y = (\prod_{i=1}^{3} \prod_{j=1}^{3} u_{ij})$$

Fixed-Point Data Types

The following diagram shows the data types used within the Matrix Product block for fixed-point signals.
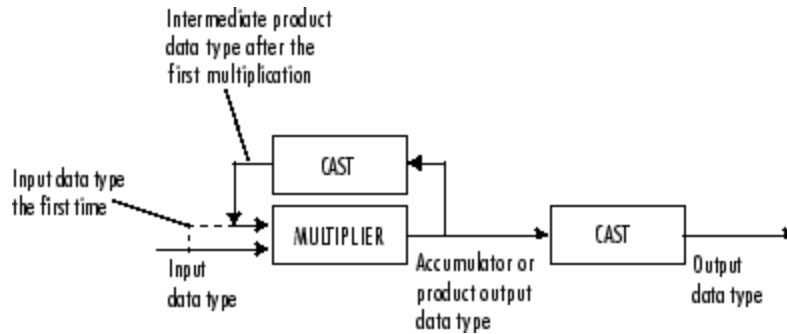
The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types". You can set the accumulator, product output, intermediate product, and output data types in the block dialog as discussed in "Dialog Box" on page 2-982 below.

Dialog Box

The **Main** pane of the Matrix Product block dialog appears as follows.

Multiply the elements of a matrix along its rows, its columns, or over all its elements. Note that 1-D input signals produce a single scalar output equal to the product of the individual elements.

Main   Data Types

Parameters
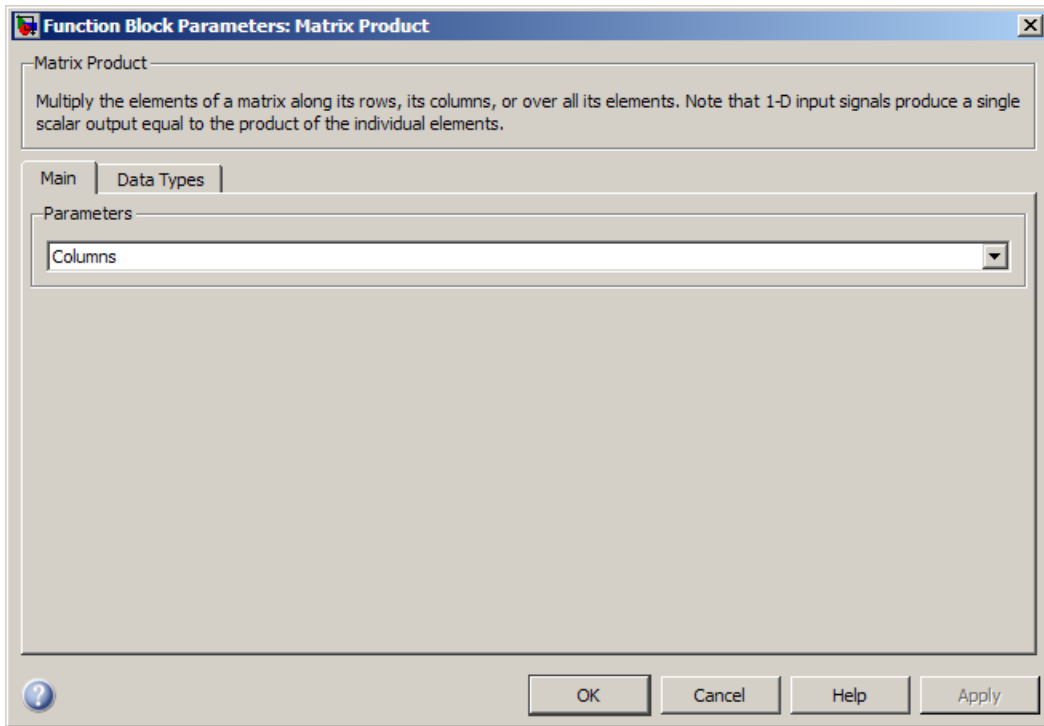
Columns

Multiply over

Indicate whether to multiply together the elements of each row, each column, or the entire input.

The **Data Types** pane of the Matrix Product block dialog appears as follows.

Matrix Product
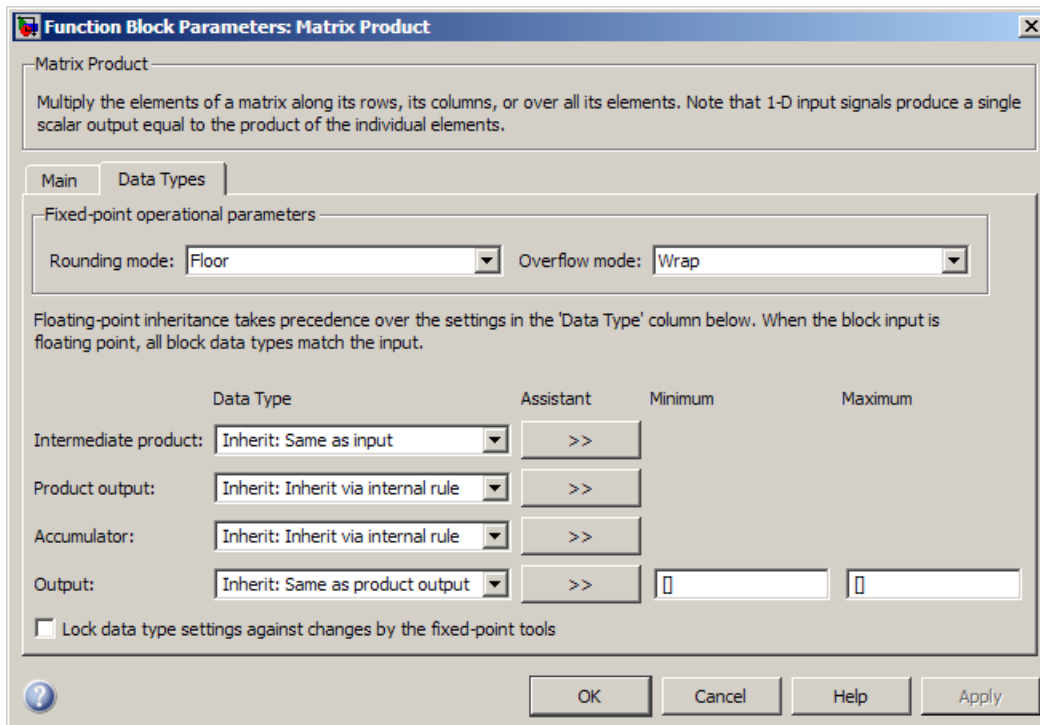
Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode
　　Select the rounding mode for fixed-point operations.
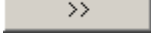
Overflow mode
　　Select the overflow mode for fixed-point operations.

Intermediate product

Specify the intermediate product data type. As shown in "Fixed-Point Data Types" on page 2-982, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. You can set it to:

- A rule that inherits a data type, for example, `Inherit:  Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button [`>>`] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-982 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button [`>>`] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Matrix Product

Accumulator data type

> Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-982 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:
>
> - A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
>
> - An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`
>
> Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.
>
> See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

> Specify the output data type. See "Fixed-Point Data Types" on page 2-982 for illustrations depicting the use of the output data type in this block. You can set it to:
>
> - A rule that inherits a data type, for example, `Inherit: Same as product output`
>
> - An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`
>
> Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.
>
> See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[ ]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[ ]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Matrix Product

Array-Vector Multiply	DSP System Toolbox
Matrix Square	DSP System Toolbox
Matrix Sum	DSP System Toolbox
prod	MATLAB

Purpose   Compute square of input matrix

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3
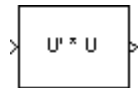
Description   The Matrix Square block computes the square of an *M*-by-*N* input
                   matrix, *u*, by premultiplying with the Hermitian transpose.

```
y = u' * u      % Equivalent MATLAB code
```

The block treats length-*M* unoriented vector inputs as an *M*-by-1
matrix. When the input is an *M*-by-*N* matrix, the output of the block is
an *N*-by-*N* matrix.

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.

Matrix Square

Dialog Box

The **Signal Attributes** pane of the Matrix Square block dialog appears as follows.
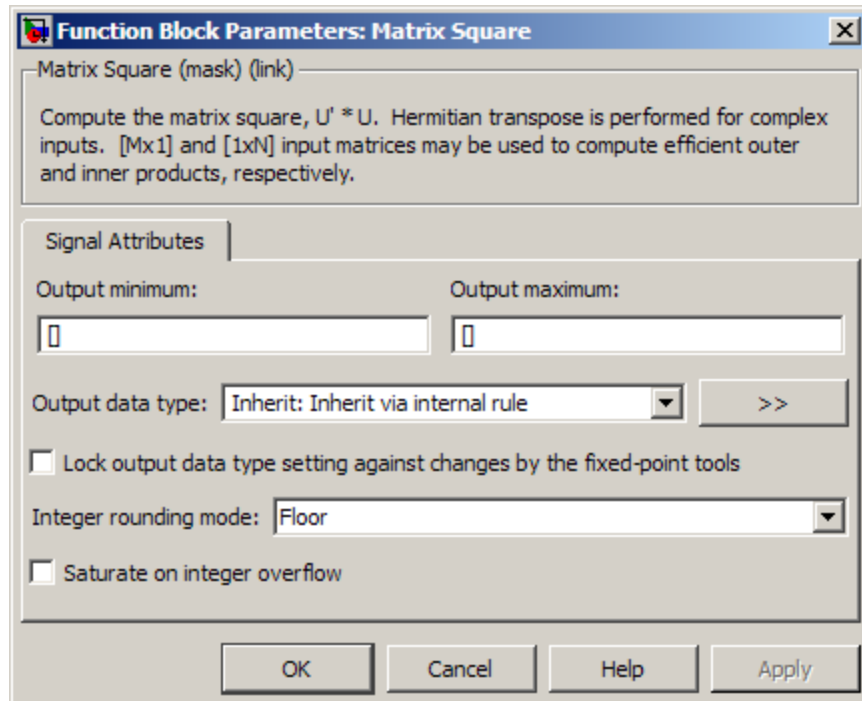
Output minimum

Specify the minimum value the block should output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Output maximum

Specify the maximum value the block should output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule

- An expression that evaluates to a valid data type, for example, fixdt([],16,0)

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Lock output data type setting against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the **Output data type** you specify on the block mask.

Integer rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select this check box to have overflows saturate to the maximum or minimum value that the data type can represent. If you clear this check box, the block wraps all overflows. See overflow mode for more information.

Matrix Square

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code

Supported Data Types

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

See Also

Matrix Multiply	DSP System Toolbox
Matrix Product	DSP System Toolbox
Matrix Sum	DSP System Toolbox
Transpose	DSP System Toolbox

Purpose   Sum matrix elements along rows, columns, or entire input

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

Description   The Matrix Sum block is an implementation of the Simulink Sum block. See Sum for more information.

Matrix Sum (Obsolete)

Purpose   Sum matrix elements along rows, columns, or entire input

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspobslib
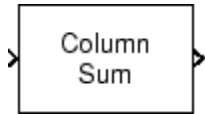
Description   The Matrix Sum block sums the elements of an M-by-N input matrix u along its rows, its columns, or over all its elements.

> Column
> Sum

When the **Sum over** 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.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \left(\sum_{j=1}^{3} u_{1j} \right) \\ \left(\sum_{j=1}^{3} u_{2j} \right) \\ \left(\sum_{j=1}^{3} u_{3j} \right) \end{bmatrix}$$

When the **Sum over** 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.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$
$$\Downarrow$$
$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} \left(\sum_{i=1}^{3} u_{i1} \right) & \left(\sum_{i=1}^{3} u_{i2} \right) & \left(\sum_{i=1}^{3} u_{i3} \right) \end{bmatrix}$$

When the **Sum over** parameter is set to Entire input, the block sums all the elements of the input together and outputs the resulting scalar.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow y = (\sum_{i=1}^{3} \sum_{j=1}^{3} u_{ij})$$

The output of the Matrix Sum block has the same frame status as the input. This block accepts real and complex fixed-point and floating-point inputs except for complex unsigned fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Matrix Sum block for fixed-point signals.
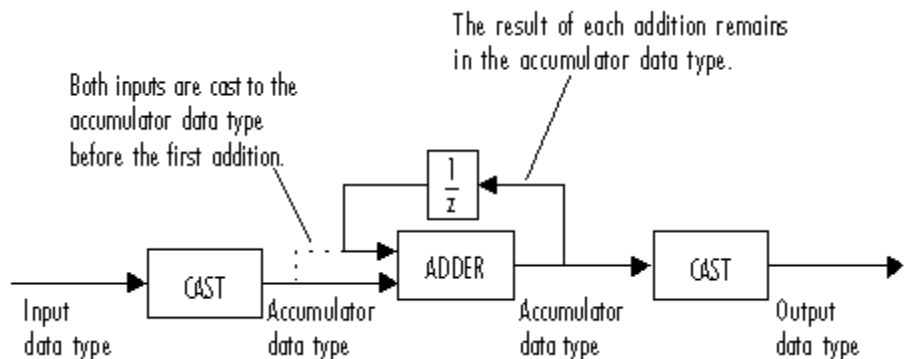
You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 2-996 below.

Matrix Sum (Obsolete)

Dialog Box

The **Main** pane of the Matrix Sum block dialog appears as follows.
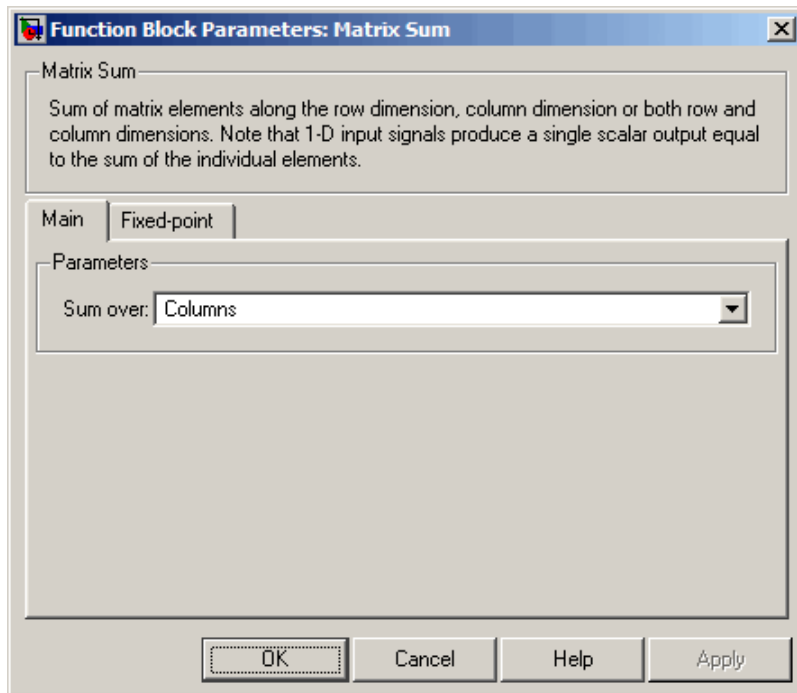
Sum over

Indicate whether to sum the elements of each row, each column, or of the entire input.

The **Fixed-point** pane of the Matrix Sum block dialog appears as follows.
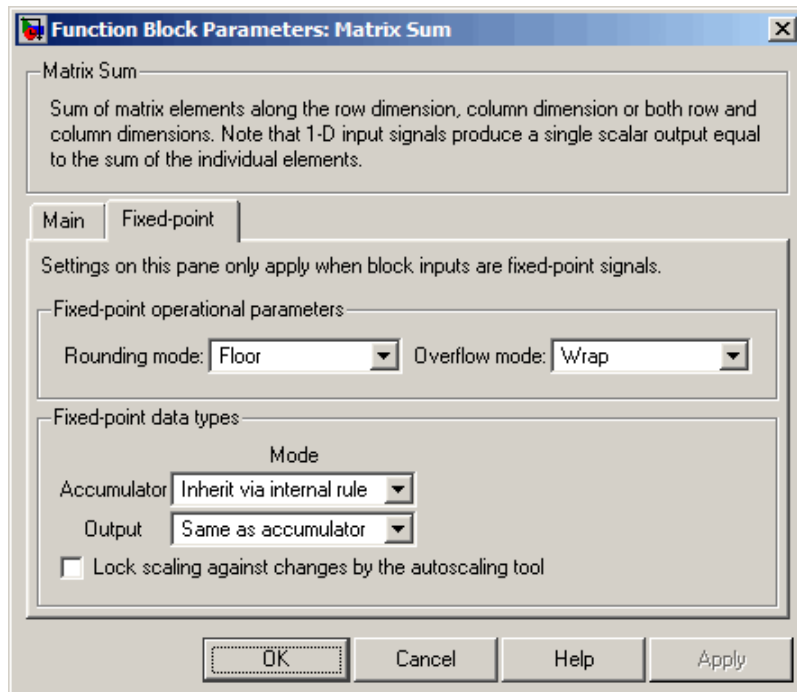
Rounding mode
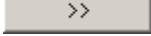Select the rounding mode for fixed-point operations.

Overflow mode
Select the overflow mode for fixed-point operations.

Matrix Sum (Obsolete)

Accumulator

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

Both inputs are cast to the
accumulator data type
before the first addition.

$\frac{1}{z}$

CAST   ADDER   CAST

Input   Accumulator   Accumulator   Output
data type   data type   data type   data type
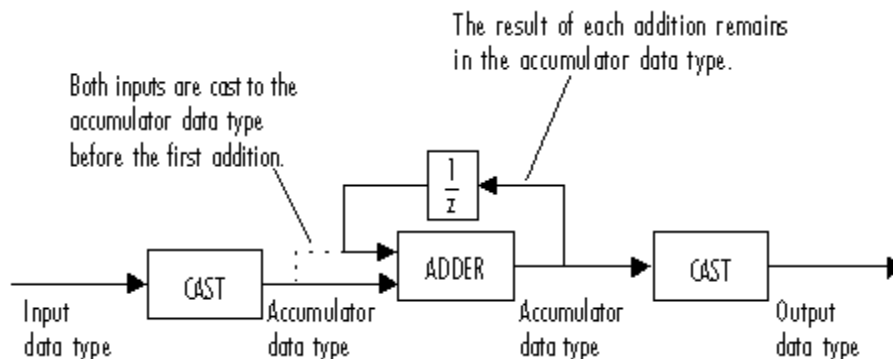
As depicted above, the elements of the block input are cast to
the accumulator data type before they are added together. The
output of the adder remains in the accumulator data type as
each element of the input is added to it. Use this parameter to
specify how you would like to designate this accumulator word
and fraction lengths:

- When you select Inherit via internal rule, the
  accumulator word length and fraction length are calculated
  automatically. For information about how the accumulator
  word and fraction lengths are calculated when an internal rule
  is used, see "Inherit via Internal Rule".

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

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

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

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

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

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

- When you select Binary point scaling, you can enter the word length and the fraction length of the output, in bits.

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

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling feature of the Fixed-Point Tool. See the fxptdlg reference page for more information.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Matrix Sum (Obsolete)

See Also

Matrix Product	DSP System Toolbox
Matrix Multiply	DSP System Toolbox
sum	MATLAB

Purpose   Display matrices as color images

Library   Sinks

dspsnks4

Description   The Matrix Viewer block displays an *M*-by-*N* matrix input by mapping
the matrix element values to a specified range of colors. The display is
updated as each new input is received. This block treats an unoriented
length *M* vector input as an *M*-by-1 matrix.

You can use the Matrix Viewer block in models running in Normal or
Accelerator simulation modes. The software does not support this block
in models running in Rapid Accelerator or External mode. For more
information about these modes, see "How Acceleration Modes Work" in
the *Simulink User's Guide*.

Image Properties

Select the **Image Properties** tab to show the image property
parameters, which control the colormap and display.

You specify the mapping of matrix element values to colors in the
Colormap matrix, **Minimum input value**, and **Maximum input
value** parameters. For a colormap with *L* colors, the colormap matrix
has dimension *L*-by-3, with one row for each color and one column for
each element of the RGB triple that defines the color. Examples of RGB
triples are

```
[ 1   0   0 ]   (red)
[ 0   0   1 ]   (blue)
[0.8 0.8 0.8]   (light gray)
```

See the `ColorSpec` property in the MATLAB documentation for
complete information about defining RGB triples.

MATLAB provides a number of functions for generating predefined
colormaps, such as `hot`, `cool`, `bone`, and `autumn`. Each of these
functions accepts the colormap size as an argument, and can be
used in the **Colormap matrix** parameter. For example, when you

specify `gray(128)` for the **Colormap matrix** parameter, the matrix is displayed in 128 shades of gray. The color in the first row of the colormap matrix represents the value specified by the **Minimum input value** parameter, and the color in the last row represents the value specified by the **Maximum input value** parameter. Values between the minimum and maximum are quantized and mapped to the intermediate rows of the colormap matrix.

The documentation for the MATLAB `colormap` function provides complete information about specifying colormap matrices, and includes a complete list of the available colormap functions.

Axis Properties

Select the **Axis Properties** tab to show the axis property parameters, which control labeling and positioning.

The **Axis origin** parameter determines where the first element of the input matrix, $U(1,1)$, is displayed. When you specify `Upper left corner`, the matrix is displayed in matrix orientation, with $U(1,1)$ in the upper-left corner.

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ U_{21} & U_{22} & U_{23} & U_{24} \\ U_{31} & U_{32} & U_{33} & U_{34} \\ U_{41} & U_{42} & U_{43} & U_{44} \end{bmatrix}$$

When you specify `Lower left corner`, the matrix is flipped vertically to image orientation, with $U(1,1)$ in the lower-left corner.

$$\begin{bmatrix} U_{41} & U_{42} & U_{43} & U_{44} \\ U_{31} & U_{32} & U_{33} & U_{34} \\ U_{21} & U_{22} & U_{23} & U_{24} \\ U_{11} & U_{12} & U_{13} & U_{14} \end{bmatrix}$$

Axis zoom, when selected, causes the image display to completely fill the figure window. Axis titles are not displayed. This option can also be

selected from the pop-up menu that is displayed when you right-click in the figure window. When **Axis zoom** is cleared, the axis labels and titles are displayed in a gray border surrounding the image axes.

Figure Window

The image title in the figure title bar is the same as the block title. The axis tick marks reflect the size of the input matrix; the x-axis is numbered from 1 to N (number of columns), and the y-axis is numbered from 1 to M (number of rows).

Right-click the image in the figure window to access the following menu items:

- **Refresh** erases all data on the scope display except for the most recent image.

- **Autoscale** recomputes the minimum and maximum input values to fit the range of values observed in a series of 10 consecutive inputs. The numerical limits selected by the autoscale feature are shown in the **Minimum input value** and **Maximum input value** parameters, where you can make further adjustments to them manually.

- **Axis zoom**, when selected, causes the image to completely fill the figure window. Axis titles are not displayed. When **Axis zoom** is cleared, the axis labels and titles are displayed in a gray border surrounding the scope axes. This option can also be set in the Axis Properties pane of the parameter dialog.

- **Colorbar**, when selected, displays a bar with the specified colormap to the right of the image axes.

- **Save Position** automatically updates the **Figure position** parameter in the **Axis Properties** pane to reflect the figure window's current position and size on the screen. To make the scope window open at a particular location on the screen when the simulation runs, drag the window to the desired location, resize it, and select **Save Position**. The parameter dialog must be closed when you select **Save Position** for the **Figure position** parameter to be updated.

Matrix Viewer

Examples   See the Spectral Analysis Using the Periodogram Method example for a demonstration of using the Matrix Viewer block to create a moving spectrogram, or time-frequency plot, of a speech signal by updating just one column of the input matrix at each sample time.

Dialog Box   The **Image Properties** pane of the Matrix Viewer block appears as follows.
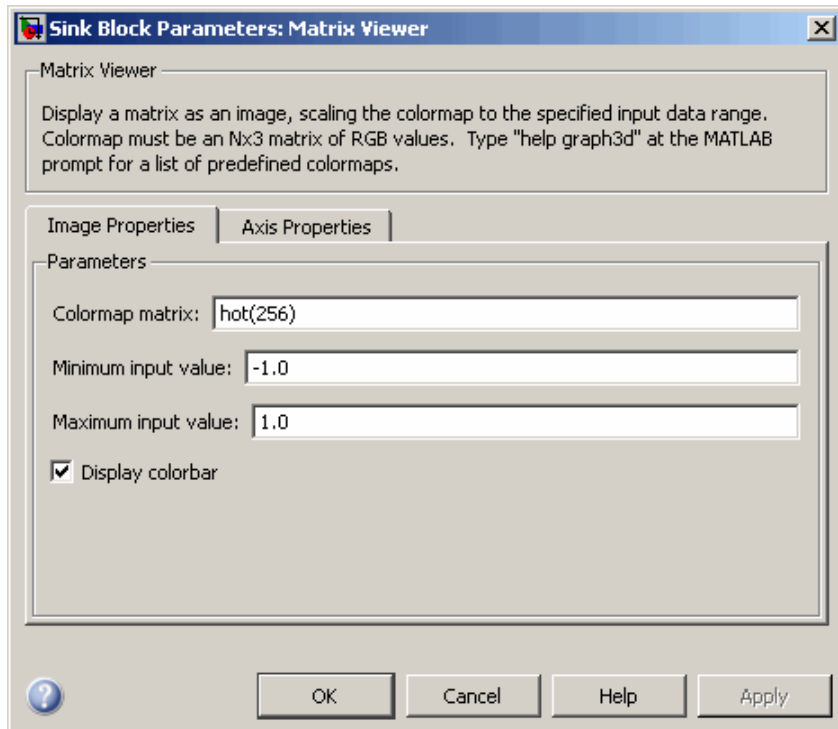
Colormap matrix

> A 3-column matrix defining the colormap as a set of RGB triples, or a call to a colormap-generating function such as hot or spring. See the ColorSpec property for complete information about

defining RGB triples, and the MATLAB `colormap` function for a list of colormap-generating functions. Tunable.

Minimum input value
>The input value to be mapped to the color defined in the first row of the colormap matrix. Right-click in the figure window and select `Autoscale` from pop-up menu to set this parameter to the minimum value observed in a series of 10 consecutive matrix inputs. Tunable.

Maximum input value
>The input value to be mapped to the color defined in the last row of the colormap matrix. Right-click in the figure window and select `Autoscale` from the pop-up menu to set this parameter to the maximum value observed in a series of 10 consecutive matrix inputs. Tunable.

Display colorbar
>Select to display a bar with the selected colormap to the right of the image axes. Tunable.

The **Axis Properties** pane of the Matrix Viewer block appears as follows.

Matrix Viewer
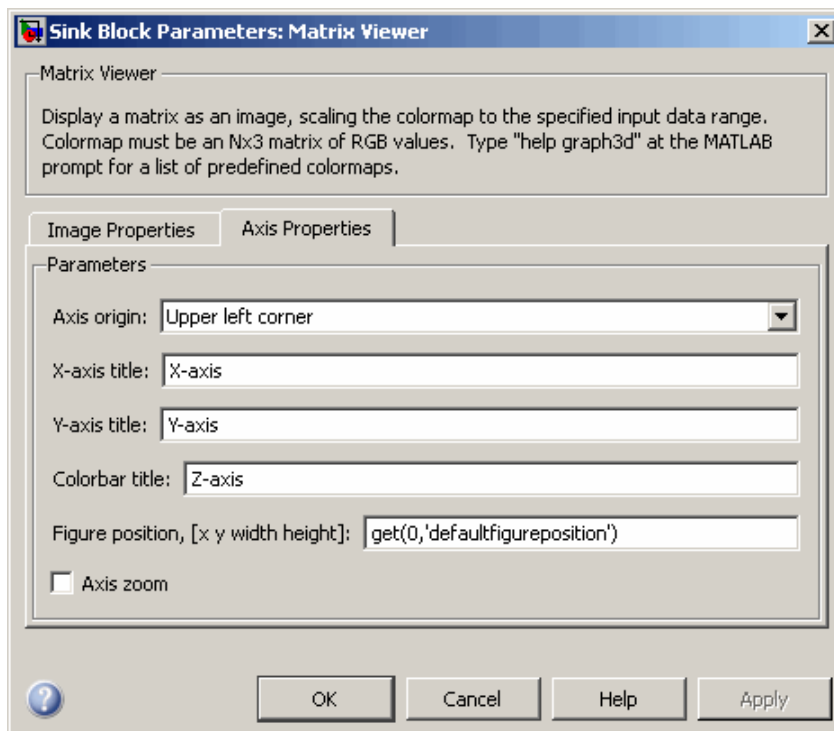
Axis origin
> The position within the axes where the first element of the input matrix, $U(1,1)$, is plotted; bottom left or top left. Tunable.

X-axis title
> The text to be displayed below the *x*-axis. Tunable.

Y-axis title
> The text to be displayed to the left of the *y*-axis. Tunable.

Colorbar title
> The text to be displayed to the right of the color bar, when **Display colorbar** is currently selected. Tunable.

Figure position, [x y width height]

A 4-element vector of the form [x y width height] specifying the position of the figure window, where (0,0) is the lower-left corner of the display. Tunable.

Axis zoom

Resizes the image to fill the figure window. Tunable.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Spectrum Scope	DSP System Toolbox
Vector Scope	DSP System Toolbox
colormap	MATLAB
ColorSpec	MATLAB
image	MATLAB

Maximum

Purpose   Find maximum values in input or sequence of inputs

Library   Statistics

               dspstat3
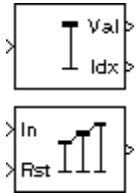
Description   The Maximum block identifies the value and/or position of the largest element in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input. The Maximum block can also track the maximum values in a sequence of inputs over a period of time. The **Mode** parameter specifies the block's mode of operation and can be set to Value, Index, Value and Index, or Running.

The Maximum block supports real and complex floating-point, fixed-point, and Boolean inputs. Real fixed-point inputs can be either signed or unsigned, while complex fixed-point inputs must be signed. The data type of the maximum values output by the block match the data type of the input. The index values output by the block are double when the input is double, and uint32 otherwise.

For the Value, Index, and Value and Index modes, the Maximum block produces identical results as the MATLAB max function when it is called as [y I] = max(u,[],D), where *u* and *y* are the input and output, respectively, *D* is the dimension, and *I* is the index.

Value Mode

When the **Mode** parameter is set to Value, the block computes the maximum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input at each sample time, and outputs the array *y*. Each element in *y* is the maximum value in the corresponding column, row, vector, or entire input. The output *y* depends on the setting of the **Find the maximum value over** parameter. For example, consider a 3-dimensional input signal of size *M*-by-*N*-by-*P*:

- Each row — The output at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the maximum value of each vector over the second dimension of the input. For an

input that is an *M*-by-*N* matrix, the output at each sample time is an *M*-by-1 column vector.

- `Each column` — The output at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the maximum value of each vector over the first dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is a 1-by-*N* row vector.

  In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- `Entire input` — The output at each sample time is a scalar that contains the maximum value in the *M*-by-*N*-by-*P* input matrix.

- `Specified dimension` — The output at each sample time depends on **Dimension**. If **Dimension** is set to `1`, the output is the same as that when you select `Each column`. If **Dimension** is set to `2`, the output is the same as when you select `Each row`. If **Dimension** is set to `3`, the output at each sample time is an *M*-by-*N* matrix containing the maximum value of each vector over the third dimension of the input.

For complex inputs, the block selects the value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input that has the maximum magnitude squared as shown below. For complex value $u = a + bi$, the magnitude squared is $a^2 + b^2$.
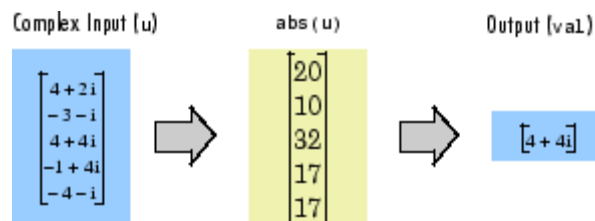
Index Mode

When **Mode** is set to `Index`, the block computes the maximum value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input, and outputs the index array *I*.

Maximum

Each element in *I* is an integer indexing the maximum value in the corresponding column, row, vector, or entire input. The output *I* depends on the setting of the **Find the maximum value over** parameter. For example, consider a 3-dimensional input signal of size *M*-by-*N*-by-*P*:

- `Each row` — The output at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the index of the maximum value of each vector over the second dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is an *M*-by-1 column vector.

- `Each column` — The output at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the index of the maximum value of each vector over the first dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is a 1-by-*N* row vector.

  In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- `Entire input` — The output at each sample time is a 1-by-3 vector that contains the location of the maximum value in the *M*-by-*N*-by-*P* input matrix. For an input that is an *M*-by-*N* matrix, the output will be a 1-by-2 vector.

- `Specified dimension` — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select `Each column`. If **Dimension** is set to 2, the output is the same as when you select `Each row`. If **Dimension** is set to 3, the output at each sample time is an *M*-by-*N* matrix containing the indices of the maximum values of each vector over the third dimension of the input.

When a maximum value occurs more than once, the computed index corresponds to the first occurrence. For example, when the input is the column vector `[3 2 1 2 3]'`, the computed one-based index of the maximum value is `1` rather than `5` when `Each column` is selected.

When inputs to the block are double-precision values, the index values are double-precision values. Otherwise, the index values are 32-bit unsigned integer values.

Value and Index Mode

When **Mode** is set to Value and Index, the block outputs both the maxima and the indices.

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. In this mode, you must also specify a value for the **Input processing** parameter:

- When you select Elements as channels (sample based), the block outputs an *M*-by-*N* array. Each element y_{ij} of the output contains the maximum value observed in element u_{ij} for all inputs since the last reset.

- When you select Columns as channels (frame based), the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the maximum value observed in the *j*th column of all inputs since the last reset, up to and including element u_{ij} of the current input.

Running Mode for Variable-Size Inputs

When your inputs are of variable size, and you set the **Mode** to Running, there are two options:

- If you set the **Input processing** parameter to Elements as channels (sample based), the state is reset.

- If you set the **Input processing** parameter to Columns as channels (frame based), then there are two cases:

  - When the input size difference is in the number of channels (i.e., number of columns), the state is reset.

  - When the input size difference is in the length of channels (i.e., number of rows), there is no reset and the running operation is carried out as usual.

Resetting the Running Maximum

The block resets the running maximum whenever a reset event is detected at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

When a reset event occurs while the **Input processing** parameter is set to Elements as channels (sample based), the running maximum for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to Columns as channels (frame based), the running maximum for each channel is initialized to the earliest value in each channel of the current input.

You specify the reset event in the **Reset port** menu:

- None — Disables the Rst port.

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

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

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

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

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

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

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

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

---

Note When running simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

Maximum

ROI Processing

To calculate the statistical value within a particular region of interest (ROI) of the input, select the **Enable ROI processing** check box. This applies to any mode other than running mode and when you set the **Find the maximum value over** parameter to `Entire input` and you select the **Enable ROI processing** check box. ROI processing applies only for 2-D inputs.

Note Full ROI processing is available only if you have a Computer Vision System Toolbox™ license. If you do not have a Computer Vision System Toolbox license, you can still use ROI processing, but are limited to the **ROI type** `Rectangles`.

Use the **ROI type** parameter to specify whether the ROI is a rectangle, line, label matrix, or binary mask. A binary mask is a binary image that enables you to specify which pixels to highlight, or select. In a label matrix, pixels equal to 0 represent the background, pixels equal to 1 represent the first object, pixels equal to 2 represent the second object, and so on. When the **ROI type** parameter is set to `Label matrix`, the Label and Label Numbers ports appear on the block. Use the Label Numbers port to specify the objects in the label matrix for which the block calculates statistics. The input to this port must be a vector of scalar values that correspond to the labeled regions in the label matrix. For more information about the format of the input to the ROI port when the ROI is a rectangle or a line, see the Draw Shapes block reference page.

For rectangular ROIs, use the **ROI portion to process** parameter to specify whether to calculate the statistical value for the entire ROI or just the ROI perimeter.

Use the **Output** parameter to specify the block output. The block can output separate statistical values for each ROI or the statistical value for all specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select `Binary mask`.

If, for the **ROI type** parameter, you select `Rectangles` or `Lines`, the **Output flag indicating if ROI is within image bounds** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

Output = Individual statistics for each ROI

Flag Port Output	Description
0	ROI is completely outside the input image.
1	ROI is completely or partially inside the input image.

Output = Single statistic for all ROIs

Flag Port Output	Description
0	All ROIs are completely outside the input image.
1	At least one ROI is completely or partially inside the input image.

If the ROI is partially outside the image, the block only computes the statistical values for the portion of the ROI that is within the image.

If, for the **ROI type** parameter, you select `Label matrix`, the **Output flag indicating if input label numbers are valid** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

Maximum

Output = Individual statistics for each ROI

Flag Port Output	Description
0	Label number is not in the label matrix.
1	Label number is in the label matrix.

Output = Single statistic for all ROIs

Flag Port Output	Description
0	None of the label numbers are in the label matrix.
1	At least one of the label numbers is in the label matrix.

Fixed-Point Data Types

The parameters on the **Data Types** pane of the block dialog are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 2-1008. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Examples   In the following ex_maximum_ref model, the Maximum block calculates the running maximum of a 3-by-2 matrix input, dsp_examples_u. The **Input processing** parameter is set to Columns as channels (frame based), so the block processes the input as a two channel signal with a frame size of three. The running maximum is reset at *t*=2 by an impulse to the block's Rst port.
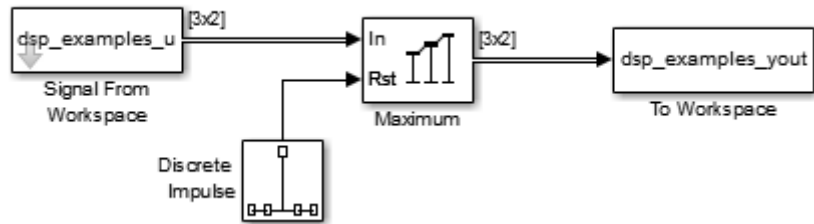
The operation of the block is shown in the following figure.
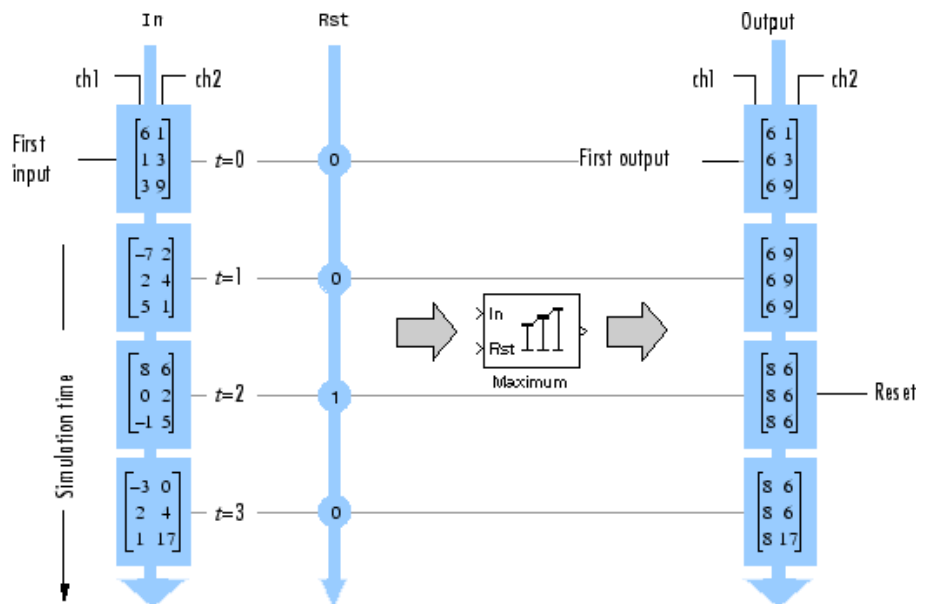
Maximum

Dialog Box

The **Main** pane of the Maximum block dialog appears as follows.
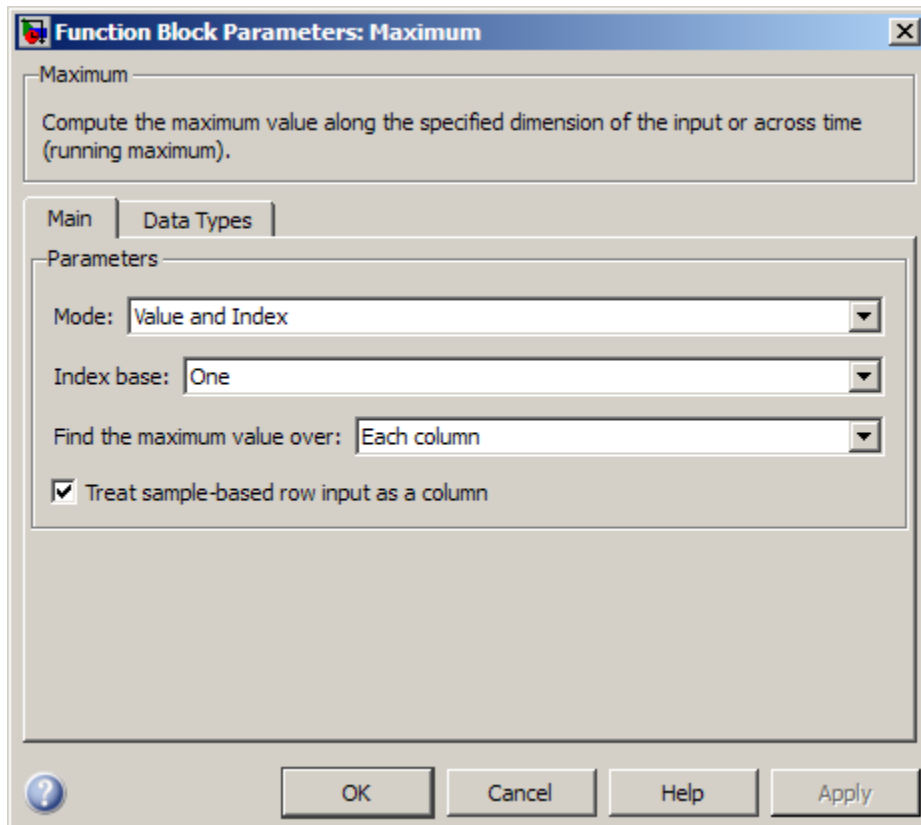
Mode

Specify the block's mode of operation:

- Value — Output the maximum value of each input

- Index — Output the index of the maximum value

- Value and index — Output both the value and the index

- `Running` — Track the maximum value of the input sequence over time

For more information, see Description.

Input processing

Specify how the block should process the input when computing the running maximum. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

This parameter appears only when you set the **Mode** to `Running`.

Note The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Index base

Specify whether the index of the maximum value is reported using one-based or zero-based numbering. This parameter is only visible when the **Mode** parameter is set to `Index` or `Value and index`.

Find the maximum value over

Specify whether to find the maximum value along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see Description.

Treat sample-based row input as a column

Select to treat sample-based length-*M* row vector inputs as *M*-by-1 column vectors. This parameter is only visible when the **Find the maximum value of** parameter is set to Each column.

Note This check box will be removed in a future release. See "Sample-Based Row Vector Processing Changes" in the DSP System Toolbox Release Notes.

Reset port

Specify the reset event that causes the block to reset the running maximum. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you set the **Mode** parameter to Running. For information about the possible values of this parameter, see "Resetting the Running Maximum" on page 2-1012.

Dimension

Specify the dimension (one-based value) of the input signal, over which the maximum is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the maximum value over** parameter is set to Specified dimension.

Enable ROI processing

Select this check box to calculate the statistical value within a particular region of each image. This parameter appears only when you set the **Find the maximum value over** parameter to Entire input, and the block is not in running mode.

Note Full ROI processing is available only if you have a Computer Vision System Toolbox license. If you do not have a Computer Vision System Toolbox license, you can still use ROI processing, but are limited to the **ROI type** Rectangles.

The ROI processing parameters appear on the dialog box as follows.
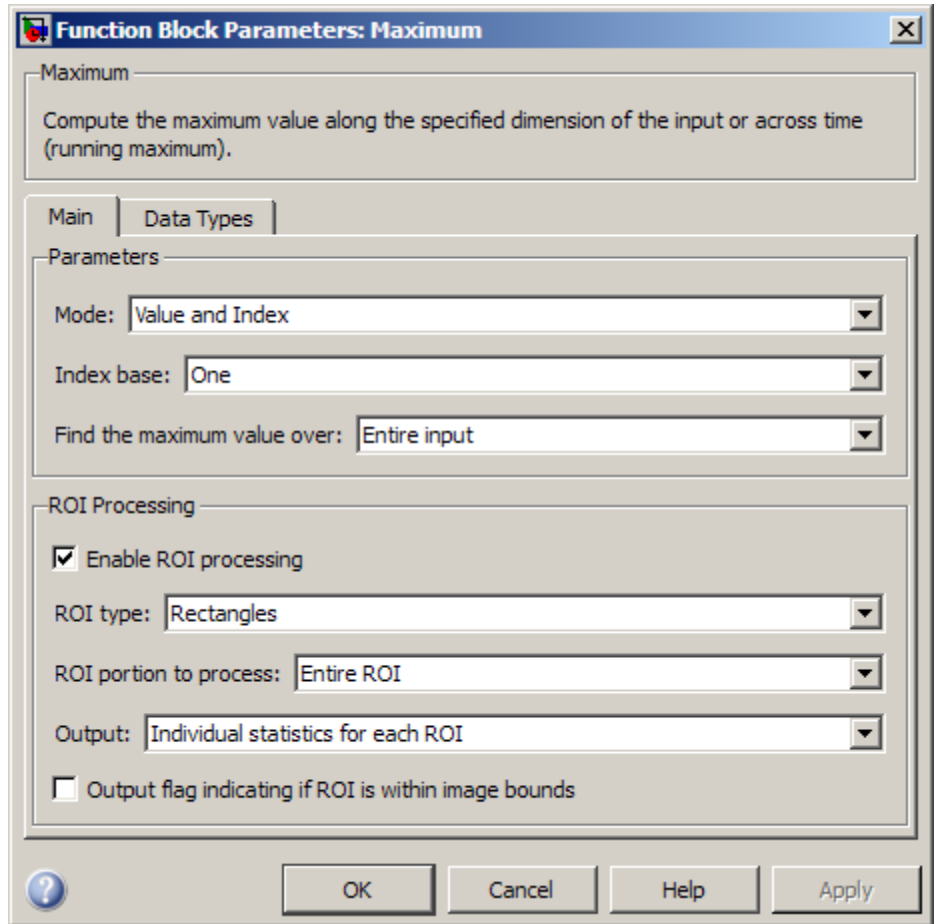
ROI type

Specify the type of ROI you want to use. Your choices are
Rectangles, Lines, Label matrix, or Binary mask.

Maximum

ROI portion to process

Specify whether you want to calculate the statistical value for the entire ROI or just the ROI perimeter. This parameter appears only if you specify an **ROI type** of Rectangles.

Output

Specify the block output. The block can output a vector of separate statistical values for each ROI or a scalar value that represents the statistical value for all the specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select Binary mask.

Output flag

☑ **Output flag indicating if ROI is within image bounds**

☑ **Output flag indicating if label numbers are valid**

When you select either of these check boxes, the Flag port appears on the block. For a description of the Flag port output, see the tables in "ROI Processing" on page 2-1014.

The **Output flag indicating if ROI is within image bounds** check box is only visible when you select Rectangles or Lines as the **ROI type**.

The **Output flag indicating if label numbers are valid** check box is only visible when you select Label matrix for the **ROI type** parameter.

The **Data Types** pane of the Maximum block dialog appears as follows.

Note The parameters on the **Data Types** pane are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 2-1008. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Maximum

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-1016 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-1016 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Reset	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Idx	• Double-precision floating point • 32-bit unsigned integers
Val	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Maximum

See Also

Mean	DSP System Toolbox
Minimum	DSP System Toolbox
MinMax	Simulink
max	MATLAB

Purpose   Find mean value of input or sequence of inputs

Library   Statistics

                     dspstat3
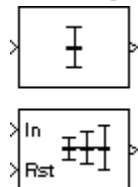
Description   The Mean block computes the mean of each row or column of the input,
                     along vectors of a specified dimension of the input, or of the entire
                     input. The Mean block can also track the mean value in a sequence of
                     inputs over a period of time. To track the mean value in a sequence of
                     inputs, select the **Running mean** check box.

Basic Operation

When you do not select the **Running mean** check box, the block
computes the mean value in each row or column of the input, along
vectors of a specified dimension of the input, or of the entire input at
each individual sample time. Each element in the output array y is the
mean value of the corresponding column, row, vector, or entire input.
The output array y depends on the setting of the **Find the mean value
over** parameter. For example, consider a 3-dimensional input signal
of size M-by-N-by-P:

- `Entire input` — The output at each sample time is a scalar that
  contains the mean value of the M-by-N-by-P input matrix.

  ```
  y = mean(u(:))      % Equivalent MATLAB code
  ```

- `Each row` — The output at each sample time consists of an
  M-by-1-by-P array, where each element contains the mean value of
  each vector over the second dimension of the input. For an input that
  is an M-by-N matrix, the output at each sample time is an M-by-1
  column vector.

  ```
  y = mean(u,2)      % Equivalent MATLAB code
  ```

- `Each column` — The output at each sample time consists of a
  1-by-N-by-P array, where each element contains the mean value of
  each vector over the first dimension of the input. For an input that

is an *M*-by-*N* matrix, the output at each sample time is a 1-by-*N* row vector.

```
y = mean(u)     % Equivalent MATLAB code
```

In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- Specified dimension — The output at each sample time depends on the value of the **Dimension** parameter. If you set the **Dimension** to 1, the output is the same as when you select Each column. If you set the **Dimension** to 2, the output is the same as when you select Each row. If you set the **Dimension** to 3, the output at each sample time is an *M*-by-*N* matrix containing the mean value of each vector over the third dimension of the input.

```
y = mean(u,Dimension)      % Equivalent MATLAB code
```

The mean of a complex input is computed independently for the real and imaginary components, as shown in the following figure.
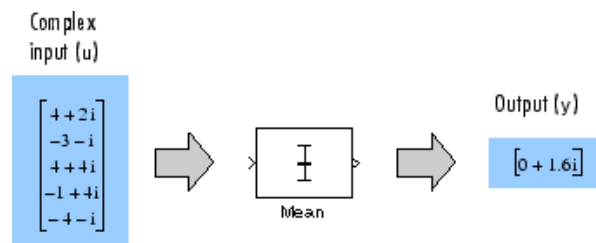
Running Operation

When you select the **Running mean** check box, the block tracks the mean value of each channel in a time sequence of inputs. In this mode, you must also specify a value for the **Input processing** parameter:

- When you select Elements as channels (sample based), the block outputs an *M*-by-*N* array. Each element y_{ij} of the output contains the mean value of the elements u_{ij} for all inputs since the last reset.

- When you select `Columns as channels (frame based)`, the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the mean of the values in the *j*th column of all inputs since the last reset, up to and including element u_{ij} of the current input.

Running Operation for Variable-Size Inputs

When your inputs are of variable size, and you select the **Running mean** check box, there are two options:

- If you set the **Input processing** parameter to `Elements as channels (sample based)`, the state is reset.

- If you set the **Input processing** parameter to `Columns as channels (frame based)`, then there are two cases:

  - When the input size difference is in the number of channels (i.e., number of columns), the state is reset.

  - When the input size difference is in the length of channels (i.e., number of rows), there is no reset and the running operation is carried out as usual.

Resetting the Running Mean

The block resets the running mean whenever a reset event is detected at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

When a reset event occurs while the **Input processing** parameter is set to `Elements as channels (sample based)`, the running mean for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to `Columns as channels (frame based)`, the running mean for each channel is initialized to the earliest value in each channel of the current input.

You specify the reset event by the **Reset port** parameter:

- `None` disables the Rst port.

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

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

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

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

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

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

Falling edge   Falling edge

Falling edge

Falling edge

Not a falling edge because it continues
a fall from a positive value to zero.

- Either edge — Triggers a reset operation when the Rst input is a
  Rising edge or Falling edge (as described earlier)

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

Note When running simulations in the Simulink MultiTasking mode,
reset signals have a one-sample latency. Therefore, when the block
detects a reset event, there is a one-sample delay at the reset port rate
before the block applies the reset. For more information on latency and
the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking
Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

ROI Processing

To calculate the statistical value within a particular region of interest
(ROI) of the input, select the **Enable ROI processing** check box.
This option is only available when the **Find the mean value over**
parameter is set to Entire input and the **Running mean** check box is
not selected. ROI processing is only supported for 2-D inputs.

Mean

Use the **ROI type** parameter to specify whether the ROI is a rectangle, line, label matrix, or binary mask. A binary mask is a binary image that enables you to specify which pixels to highlight, or select. In a label matrix, pixels equal to 0 represent the background, pixels equal to 1 represent the first object, pixels equal to 2 represent the second object, and so on. When the **ROI type** parameter is set to Label matrix, the Label and Label Numbers ports appear on the block. Use the Label Numbers port to specify the objects in the label matrix for which the block calculates statistics. The input to this port must be a vector of scalar values that correspond to the labeled regions in the label matrix. For more information about the format of the input to the ROI port when the ROI is a rectangle or a line, see the Draw Shapes block reference page.

For rectangular ROIs, use the **ROI portion to process** parameter to specify whether to calculate the statistical value for the entire ROI or just the ROI perimeter.

Use the **Output** parameter to specify the block output. The block can output separate statistical values for each ROI or the statistical value for all specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select Binary mask.

If, for the **ROI type** parameter, you select Rectangles or Lines, the **Output flag indicating if ROI is within image bounds** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

Output = Individual statistics for each ROI

Flag Port Output	Description
0	ROI is completely outside the input image.
1	ROI is completely or partially inside the input image.

Output = Single statistic for all ROIs

Flag Port Output	Description
0	All ROIs are completely outside the input image.
1	At least one ROI is completely or partially inside the input image.

If the ROI is partially outside the image, the block only computes the statistical values for the portion of the ROI that is within the image.

If, for the **ROI type** parameter, you select Label matrix, the **Output flag indicating if input label numbers are valid** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

Output = Individual statistics for each ROI

Flag Port Output	Description
0	Label number is not in the label matrix.
1	Label number is in the label matrix.

Mean

Output = Single statistic for all ROIs

Flag Port Output	Description
0	None of the label numbers are in the label matrix.
1	At least one of the label numbers is in the label matrix.

Fixed-Point Data Types

The following diagram shows the data types used within the Mean block for fixed-point signals.
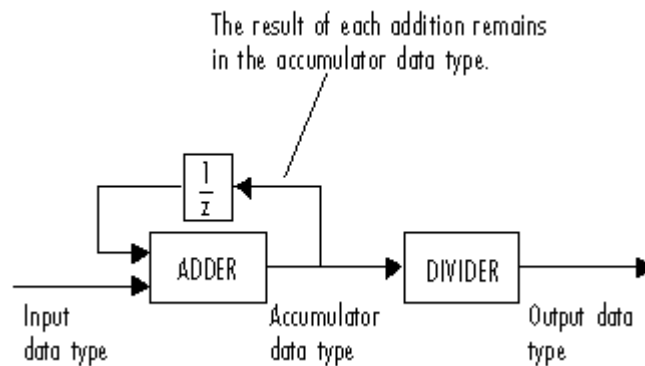
You can set the accumulator and output data types in the block dialog, as discussed in "Dialog Box" on page 2-1035.

Examples   In the ex_mean_ref model, the Mean block calculates the running mean of a 3-by-2 matrix input, u. The **Input processing** parameter is set to Columns as channels (frame based), so the block processes the input as a two channel signal with a frame size of three. The running mean is reset at t=2 by an impulse to the block's Rst port.
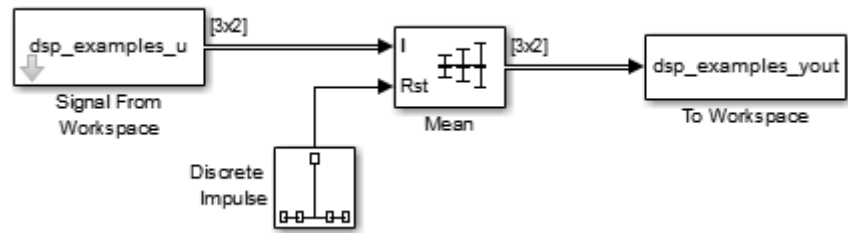
The operation of the block is shown in the following figure.
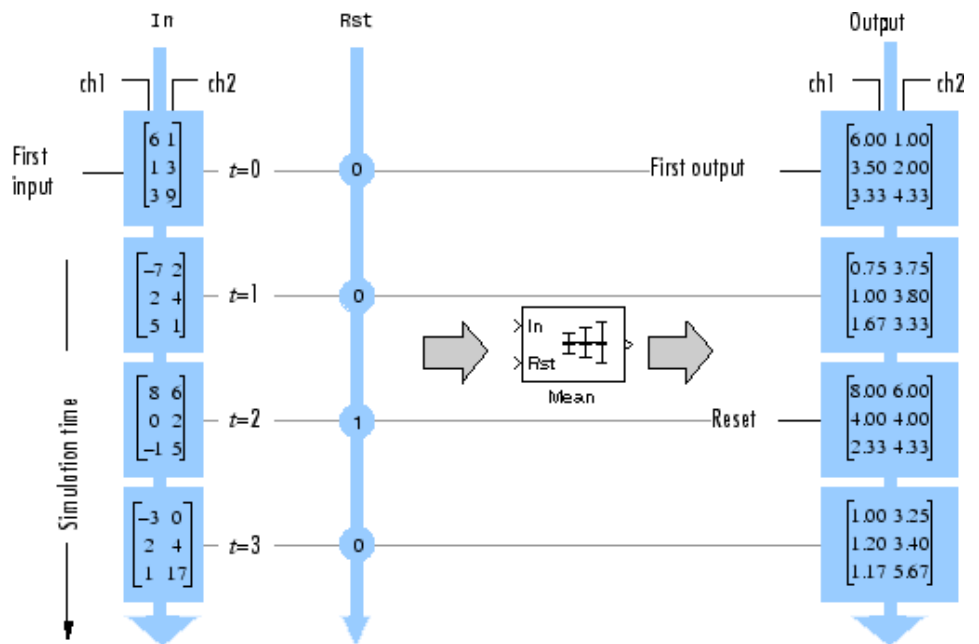
Dialog Box

The **Main** pane of the Mean block dialog appears as follows.

Mean
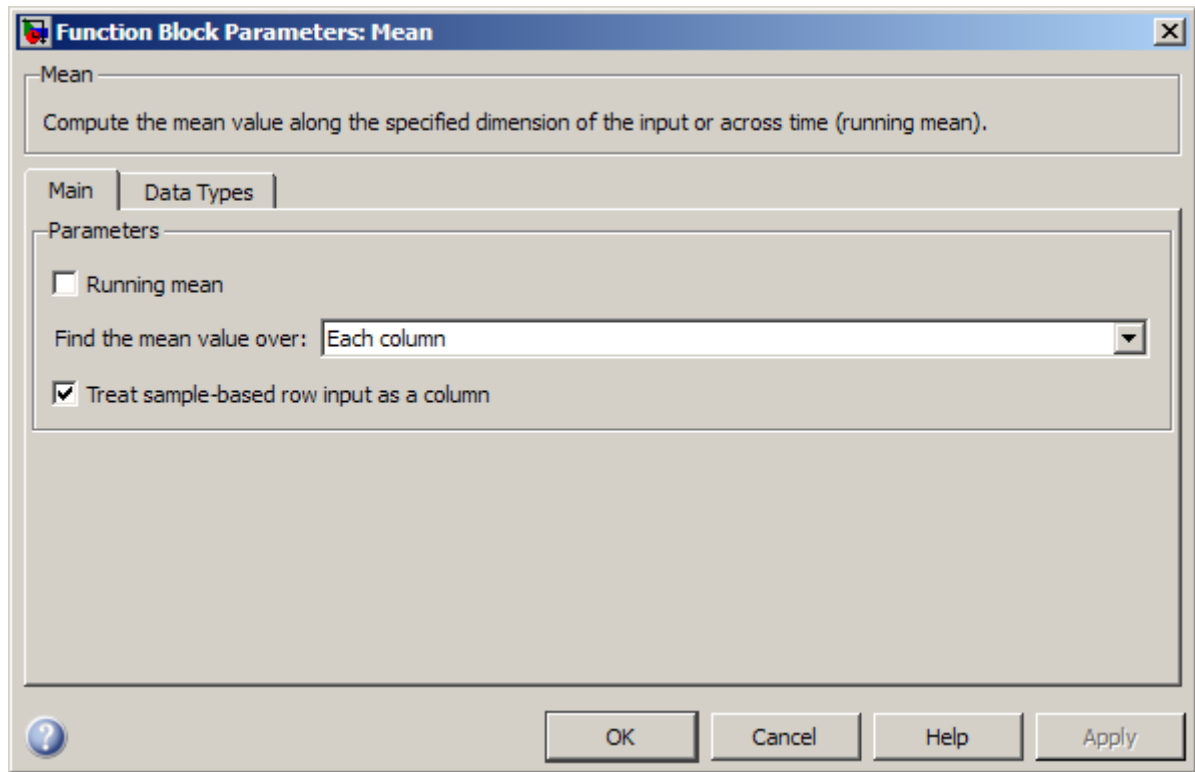
Running mean

Enables running operation when selected.

Input processing

Specify how the block should process the input when computing the running mean. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

This parameter appears only when you select the **Running mean** check box.

Note The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Reset port

Specify the reset event that causes the block to reset the running mean. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you select the **Running mean** check box. For more information, see "Resetting the Running Mean" on page 2-1029.

Find the mean value over

Specify whether to find the mean value along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see "Basic Operation" on page 2-1027.

Treat sample-based row input as a column

Select to treat sample-based length-M row vector inputs as M-by-1 column vectors. This parameter is only visible when the **Find the mean value over** parameter is set to `Each column`.

Note This check box will be removed in a future release. See "Sample-Based Row Vector Processing Changes" in the DSP System Toolbox Release Notes.

Mean

Dimension

Specify the dimension (one-based value) of the input signal, over which the mean is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the mean value over** parameter is set to `Specified dimension`.

Enable ROI Processing

Select this check box to calculate the statistical value within a particular region of each image. This parameter is only available when the **Find the mean value over** parameter is set to `Entire input`, and the block is not in running mode.

Note Full ROI processing is available only if you have a Computer Vision System Toolbox license. If you do not have a Computer Vision System Toolbox license, you can still use ROI processing, but are limited to the **ROI type** `Rectangles`.

ROI type

Specify the type of ROI you want to use. Your choices are `Rectangles`, `Lines`, `Label matrix`, or `Binary mask`.

ROI portion to process

Specify whether you want to calculate the statistical value for the entire ROI or just the ROI perimeter. This parameter is only visible if, for the **ROI type** parameter, you specify `Rectangles`.

Output

Specify the block output. The block can output a vector of separate statistical values for each ROI or a scalar value that represents the statistical value for all the specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select `Binary mask`.

Output flag

☑ **Output flag indicating if ROI is within image bounds**

☑ **Output flag indicating if label numbers are valid**

When you select either of these check boxes, the Flag port appears on the block. For a description of the Flag port output, see the tables in "ROI Processing" on page 2-1031.

The **Output flag indicating if ROI is within image bounds** check box is only visible when you select Rectangles or Lines as the **ROI type**.

The **Output flag indicating if label numbers are valid** check box is only visible when you select Label matrix for the **ROI type** parameter.

The **Data Types** pane of the Mean block dialog appears as follows.

Rounding mode
Select the rounding mode for fixed-point operations.

Overflow mode
Select the overflow mode for fixed-point operations.

Accumulator data type
Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-1034 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, Inherit: Same as input

• An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-1034 for illustrations depicting the use of the output data type in this block. You can set it to:

• A rule that inherits a data type, for example, `Inherit:   Same as accumulator`

• An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

• Simulation range checking (see "Signal Ranges")

• Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Reset	• Double-precision floating point
	• Single-precision floating point
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
ROI	Rectangles and lines: • Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Binary Mask: • Boolean
Label	• 8-, 16-, and 32-bit unsigned integers
Label Numbers	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Flag	• Boolean

See Also

Maximum	DSP System Toolbox
Median	DSP System Toolbox
Minimum	DSP System Toolbox
Standard Deviation	DSP System Toolbox
mean	MATLAB

Median

Purpose	Find median value of input
Library	Statistics
	dspstat3

Description
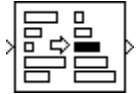
The Median block computes the median value of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input. The median of a set of input values is calculated as follows:

1 The values are sorted.

2 If the number of values is odd, the median is the middle value.

3 If the number of values is even, the median is the average of the two middle values.

For a given input u, the size of the output array y depends on the setting of the **Find the median value over** parameter. For example, consider a 3-dimensional input signal of size M-by-N-by-P:

- Entire input — The output at each sample time is a scalar that contains the median value of the M-by-N-by-P input matrix.

  ```
  y = median(u(:))     % Equivalent MATLAB code
  ```

- Each row — The output at each sample time consists of an M-by-1-by-P array, where each element contains the median value of each vector over the second dimension of the input. For an input that is an M-by-N matrix, the output is an M-by-1 column vector.

  ```
  y = median(u,2)      % Equivalent MATLAB code
  ```

- Each column — The output at each sample time consists of a 1-by-N-by-P array, where each element contains the median value of each vector over the first dimension of the input. For an input that is an M-by-N matrix, the output at each sample time is a 1-by-N row vector.

```
y = median(u)      % Equivalent MATLAB code
```

For convenience, length-*M* 1-D vector inputs are treated as *M*-by-1 column vectors when the block is in this mode. Sample-based length-*M* row vector inputs are also treated as *M*-by-1 column vectors when the **Treat sample-based row input as a column** check box is selected.

- Specified dimension — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select Each column. If **Dimension** is set to 2, the output is the same as when you select Each row. If **Dimension** is set to 3, the output at each sample time is an *M*-by-*N* matrix containing the median value of each vector over the third dimension of the input.

```
y = median(u,Dimension)      % Equivalent MATLAB code
```

The block sorts complex inputs according to their magnitude.

Fixed-Point Data Types

For fixed-point inputs, you can specify accumulator, product output, and output data types as discussed in "Dialog Box" on page 2-1046. Not all these fixed-point parameters are applicable for all types of fixed-point inputs. The following table shows when each kind of data type and scaling is used.

	Output data type	Accumulator data type	Product output data type
Even *M*	X	X	
Odd *M*	X		
Odd *M* and complex	X	X	X
Even *M* and complex	X	X	X

Median

The accumulator and output data types and scalings are used for fixed-point signals when *M* is even. The result of the sum performed while calculating the average of the two central rows of the input matrix is stored in the accumulator data type and scaling. The total result of the average is then put into the output data type and scaling.

The accumulator and product output parameters are used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before the input elements are sorted, as described in Description. The results of the squares of the real and imaginary parts are placed into the product output data type and scaling. The result of the sum of the squares is placed into the accumulator data type and scaling.

For fixed-point inputs that are both complex and have even *M*, the data types are used in all of the ways described. Therefore, in such cases, the accumulator type is used in two different ways.

Dialog Box

The **Main** pane of the Median block dialog appears as follows.

Function Block Parameters: Median ✕

Median

Compute the median value along the specified dimension of the input or across time (running median).

The 'Product output' and 'Accumulator' parameters apply only for complex fixed-point inputs.

Main | Data Types

Parameters

Sort algorithm: Quick sort ▾

Find the median value over: Each column ▾

☑ Treat sample-based row input as a column

⊘ | OK | Cancel | Help | Apply
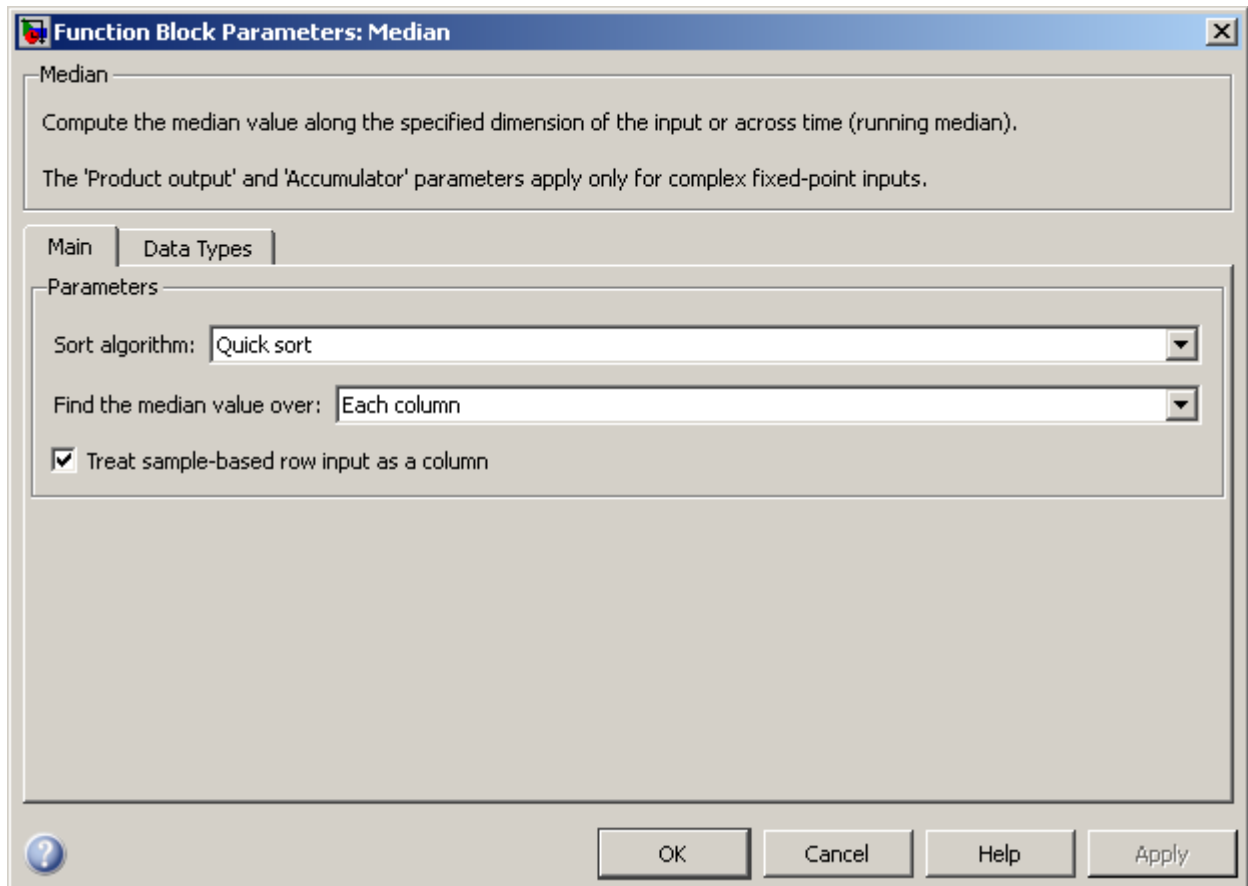
Sort algorithm
Specify whether to sort the elements of the input using a `Quick sort` or an `Insertion sort` algorithm.

Find the median value over
Specify whether to find the median value along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see Description.

Treat sample-based row input as a column
Select to treat sample-based length-*M* row vector inputs as *M*-by-1 column vectors. This parameter is only visible when the **Find the median value over** parameter is set to Each column.

Note This check box will be removed in a future release. See "Sample-Based Row Vector Processing Changes" in the DSP System Toolbox Release Notes.

Dimension
Specify the dimension (one-based value) of the input signal, over which the median is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the median value over** parameter is set to Specified dimension.

The **Data Types** pane of the Median block dialog appears as follows.
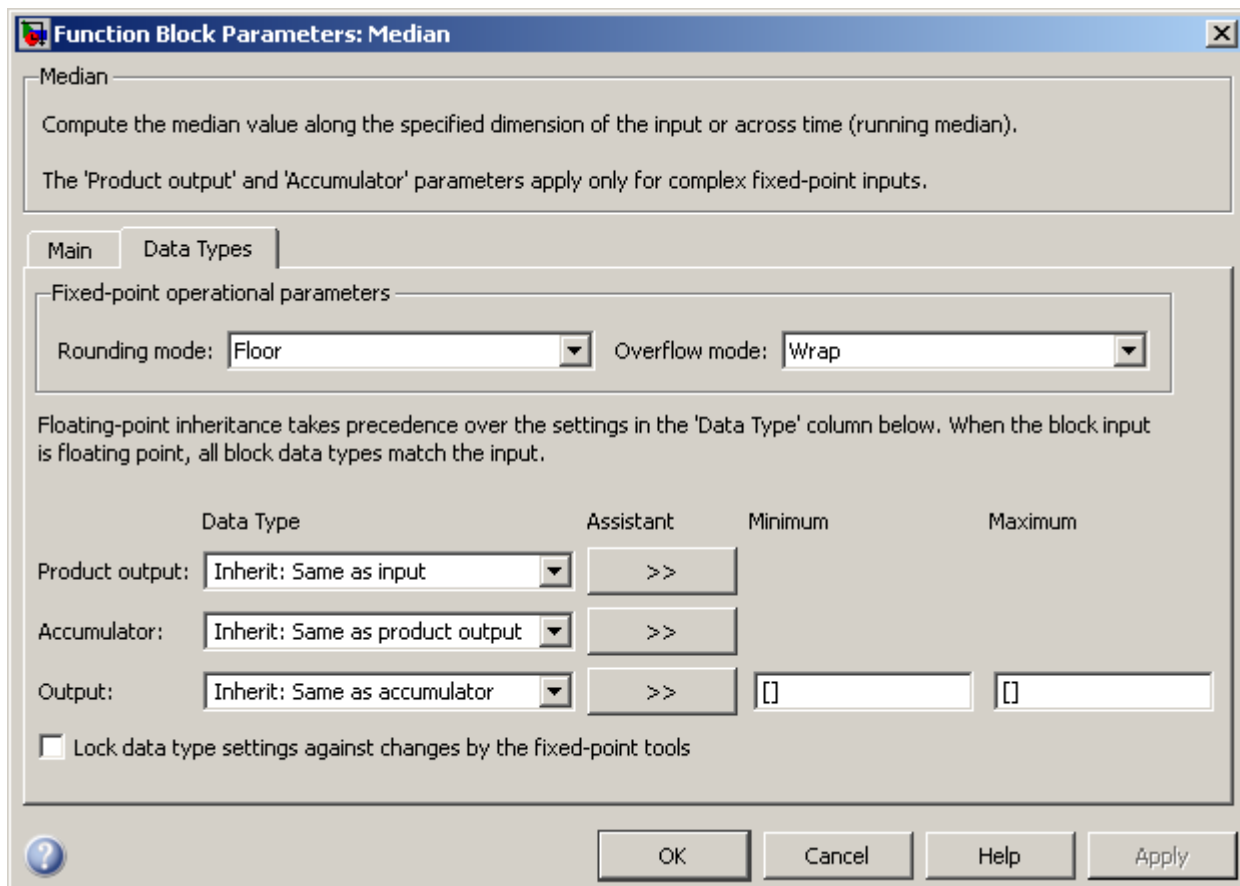
Function Block Parameters: Median ✕

─ Median ───────────────────────────────────────

Compute the median value along the specified dimension of the input or across time (running median).

The 'Product output' and 'Accumulator' parameters apply only for complex fixed-point inputs.

| Main | Data Types |

─ Fixed-point operational parameters ─────────────────────────

Rounding mode: [Floor ▾]   Overflow mode: [Wrap ▾]

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.

	Data Type	Assistant	Minimum	Maximum
Product output:	Inherit: Same as input ▾	>>		
Accumulator:	Inherit: Same as product output ▾	>>		
Output:	Inherit: Same as accumulator ▾	>>	[]	[]

☐ Lock data type settings against changes by the fixed-point tools

ⓘ   [OK] [Cancel] [Help] [Apply]

Note Floating-point inheritance takes precedence over the data type settings defined on this pane. When inputs are floating point, the block ignores these settings, and all internal data types are floating point.

Rounding mode
    Select the rounding mode for fixed-point operations.

Median

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-1045 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-1045 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-1045 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Median

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, 32-, and 128-bit signed integers • 8-, 16-, 32-, and 128-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, 32-, and 128-bit signed integers • 8-, 16-, 32-, and 128-bit unsigned integers

See Also

Maximum	DSP System Toolbox
Mean	DSP System Toolbox
Minimum	DSP System Toolbox
Sort	DSP System Toolbox
Standard Deviation	DSP System Toolbox
Variance	DSP System Toolbox
median	MATLAB

Purpose   Output values from controls on MIDI control surface

Library   Sources

       dspsrcs4

Description The MIDI Controls block outputs values from controls on a MIDI control surface in real time.

> MIDI Controls

Use the **MIDI device** parameter to specify the name of the MIDI control surface device from which to receive control values. You can choose:

- Default

- Specify other

If you choose Default, the block looks for a MATLAB preference with a group named midi and preference named DefaultDevice. You can set this preference using the MATLAB setpref function. For example, if the desired device is named BCF2000, you can type the following command at the MATLAB command line:

```
>> setpref('midi', 'DefaultDevice', 'BCF2000');
```

If the block does not find this preference, it then attempts to choose a device using an algorithm that is unspecified and platform dependent.

If you choose Specify other, then a **MIDI device name** edit box appears for you to enter a MATLAB expression for the device name. Enter any MATLAB expression that can evaluate to a string. Literal names must be enclosed in quotes, (for example, 'BCF2000').

You can determine the name of your MIDI device using the MATLAB function midiid, discussed in "Identifying MIDI Device Names and Control Numbers" on page 2-1055.

Use the **MIDI controls** parameter to specify the controls on the MIDI device to which the block should respond. This parameter also determines the size of the block output port. You can choose:

- `Respond to any control`

- `Respond to specified controls`

If you choose `Respond to any control`, then the block output will be a scalar. This scalar outputs the value from any and all controls that are manipulated on the MIDI device. Use this option in simple cases when you need only a single control value and the control to which it responds is unimportant.

If you choose `Respond to specified controls`, then a **MIDI control numbers** edit box opens. In this box, enter a MATLAB expression for the device control numbers. Enter any MATLAB expression that can evaluate to a row vector of real double-precision values. The block outputs a 1-D vector with one element corresponding to the output of each specified control.

Use the **Initial values** parameter to specify the value of the block output when simulation starts. The MIDI protocol transmits control values only when a control changes. This protocol provides no means for the block to query the current value of a control. Thus, the block must have some initial value to output until it receives a control change from the device.

Use the **Send initial values to device at start** check box to synchronize the device controls with the block outputs when simulation starts. Some MIDI control surfaces are bidirectional, meaning that they not only send control values but can also receive them. For example, some devices have motorized controls that move to the appropriate position when they receive a control value. If you have such a bidirectional device, select this check box. The block attempts to send the initial values to the device when the simulation starts. No diagnostic message appears if the attempt fails.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy. However you must account for these extra .dll files when doing so. The packNGo function creates a single .zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

Output Port

The MIDI Controls block output is a vector whose width is determined by the **MIDI controls** and **MIDI control numbers** parameters previously described. The output data type is real double-precision floating point. The output values range from 0.0 to 1.0, inclusively. The output port back inherits its sample time.

Identifying MIDI Device Names and Control Numbers

To specify a particular control on a particular MIDI device, you must know the name assigned to the device by the operating system. In addition, a number is always associated with the control. You can interactively discover this information using the MATLAB function, midiid. Follow these steps to identify device names and control numbers:

1 Verify that MIDI control surface device is correctly connected to the host computer running MATLAB.

Note For the most consistent behavior, MathWorks recommends that you connect your MIDI control surface device to your computer before starting MATLAB. In some circumstances MATLAB may not be able to find your device if you connect it after starting your MATLAB session. Also, it may not find your device if you disconnect it and reconnect it during your MATLAB session.

---

2 Type the following command at the MATLAB command line.

```
>> [ctlnum devname] = midiid
```

You are prompted to move the control in which you are interested.

```
>> [ctlnum devname] = midiid
Move the control you wish to identify; type ^C to abort.
Waiting for control message ...
```

3 Move the control. `midiid` detects the movement and returns the device name and control number.

```
>> [ctlnum devname] = midiid
Move the control you wish to identify; type ^C to abort.
Waiting for control message ... done
ctlnum =
        1081
devname =
        BCF2000
>>
```

4 Use the device name in the block dialog, or set it as the default device using `setpref`. Then, enter the control number in the block dialog. Concatenate the number with other control numbers as needed.

Examples

How to Output Values from Controls

Use this example to familiarize yourself with how to set controls in the MIDI Controls block as it interacts with the MIDI control surface.

Open the ex_simplemidi model, and follow these steps:

1

Connect a MIDI device to the computer.

2

Use `midiid` to determine the name of the device, and set it on the MIDI Controls block.

3

Verify that any control changes the display value.

4

Use `midiid` to determine the number of a particular control, and set that on the MIDI Controls block.

5

Verify that a particular control changes the display value and that other controls do not.

6

Use `midiid` to determine the number of a few more controls, and set those on the MIDI Controls block.

7

Verify that the display block shows the correct number of values. Also verify that the controls you specified change the appropriate display values and that the other controls do not change the values.

8

Set each control to have a unique initial value.

9

Verify that the correct initial values appear on the display when the model starts.

10

If your MIDI device is bidirectional, on the MIDI Controls block, select the **Send initial values to device at start** check box.

11

Verify that the controls are set to the correct initial values when the model starts.

MIDI Controls
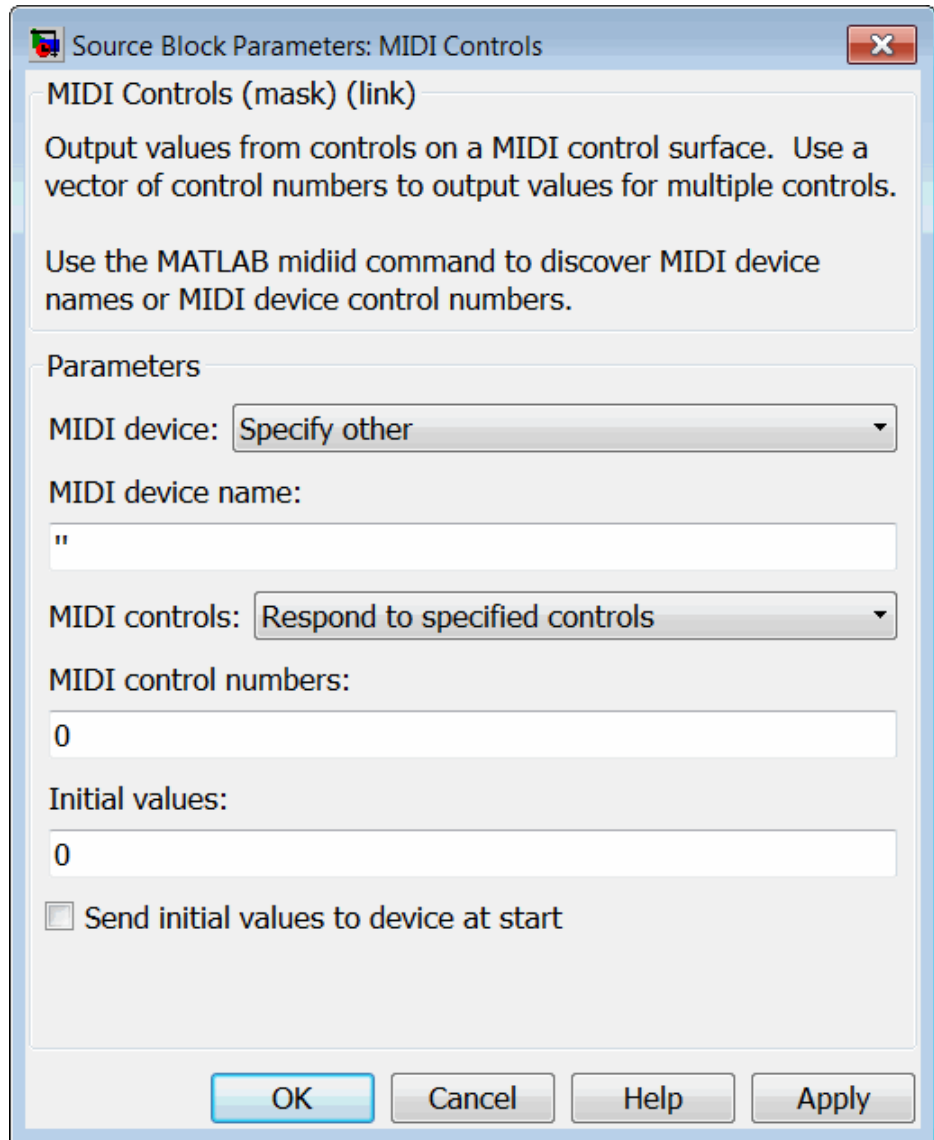
Dialog Box

MIDI device

Specify whether to use a default MIDI device, or specify a particular device by name.

MIDI device name

Specify the name of a particular MIDI control surface device from which to receive control values.

MIDI controls

Specify whether to respond to any control on the MIDI device or respond to particular specified controls.

MIDI control numbers

Specify particular controls to which the block should respond.

Initial values

Specify initial values to output when simulation starts.

Send initial values to device at start

Select this check box to attempt to synchronize a bidirectional MIDI device with block initial values when simulation starts.

Supported Data Types

Port	Supported Data Types
Output	• Double-precision floating point

Minimum

Purpose   Find minimum values in input or sequence of inputs

Library   Statistics

dspstat3
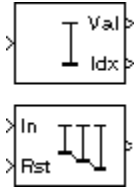
Description   The Minimum block identifies the value and/or position of the smallest
element in each row or column of the input, along vectors of a specified
dimension of the input, or of the entire input. The Minimum block can
also track the minimum values in a sequence of inputs over a period of
time. The **Mode** parameter specifies the block's mode of operation, and
can be set to Value, Index, Value and Index, or Running.

The Minimum block supports real and complex floating-point,
fixed-point, and Boolean inputs. Real fixed-point inputs can be either
signed or unsigned, while complex fixed-point inputs must be signed.
The data type of the minimum values output by the block match the
data type of the input. The index values output by the block are double
when the input is double, and uint32 otherwise.

For the Value, Index, and Value and Index modes, the Minimum
block produces identical results as the MATLAB min function when it
is called as [y I] = min(u,[],D), where *u* and *y* are the input and
output, respectively, *D* is the dimension, and *I* is the index.

Value Mode

When the **Mode** parameter is set to Value, the block computes the
minimum value in each row or column of the input, along vectors of a
specified dimension of the input, or of the entire input at each sample
time, and outputs the array *y*. Each element in *y* is the minimum
value in the corresponding column, row, vector, or entire input. The
output *y* depends on the setting of the **Find the minimum value
over** parameter. For example, consider a 3-dimensional input signal
of size *M*-by-*N*-by-*P*:

- Each row — The output at each sample time consists of an
  M-by-1-by-*P* array, where each element contains the minimum value
  of each vector over the second dimension of the input. For an input

that is an *M*-by-*N* matrix, the output at each sample time is an *M*-by-1 column vector.

- `Each column` — The output at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the minimum value of each vector over the first dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is a 1-by-*N* row vector.

  In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- `Entire input` — The output at each sample time is a scalar that contains the minimum value in the *M*-by-*N*-by-*P* input matrix.

- `Specified dimension` — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select `Each column`. If **Dimension** is set to 2, the output is the same as when you select `Each row`. If **Dimension** is set to 3, the output at each sample time is an *M*-by-*N* matrix containing the minimum value of each vector over the third dimension of the input.

For complex inputs, the block selects the value in each row or column of the input, along vectors of a specified dimension of the input, or of the entire input that has the minimum magnitude squared as shown below.

For complex value $u = a + bi$, the magnitude squared is $a^2 + b^2$.
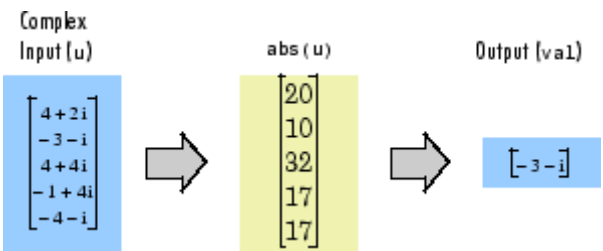
Index Mode

When **Mode** is set to `Index`, the block computes the minimum value in each row or column of the input, along vectors of a specified dimension

of the input, or of the entire input, and outputs the index array *I*. Each element in *I* is an integer indexing the minimum value in the corresponding column, row, vector, or entire input. The output *I* depends on the setting of the **Find the minimum value over** parameter. For example, consider a 3-dimensional input signal of size *M*-by-*N*-by-*P*:

- `Each row` — The output at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the index of the minimum value of each vector over the second dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is an *M*-by-1 column vector.

- `Each column` — The output at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the index of the minimum value of each vector over the first dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is a 1-by-*N* row vector.

  In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- `Entire input` — The output at each sample time is a 1-by-3 vector that contains the location of the minimum value in the *M*-by-*N*-by-*P* input matrix. For an input that is an *M*-by-*N* matrix, the output will be a 1-by-2 vector.

- `Specified dimension` — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select `Each column`. If **Dimension** is set to 2, the output is the same as when you select `Each row`. If **Dimension** is set to 3, the output at each sample time is an *M*-by-*N* matrix containing the indices of the minimum values of each vector over the third dimension of the input.

When a minimum value occurs more than once, the computed index corresponds to the first occurrence. For example, when the input is the column vector `[-1 2 3 2 -1]'`, the computed one-based index of the minimum value is `1` rather than `5` when `Each column` is selected.

Value and Index Mode

When **Mode** is set to `Value and Index`, the block outputs both the minima and the indices.

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. In this mode, you must also specify a value for the **Input processing** parameter:

- When you select `Elements as channels (sample based)`, the block outputs an *M*-by-*N* array. Each element y_{ij} of the output contains the minimum value observed in element u_{ij} for all inputs since the last reset.

- When you select `Columns as channels (frame based)`, the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the minimum value observed in the *j*th column of all inputs since the last reset, up to and including element u_{ij} of the current input.

Running Mode for Variable-Size Inputs

When your inputs are of variable size, and you set the **Mode** to `Running`, there are two options:

- If you set the **Input processing** parameter to `Elements as channels (sample based)`, the state is reset.

- If you set the **Input processing** parameter to `Columns as channels (frame based)`, then there are two cases:

  - When the input size difference is in the number of channels (i.e., number of columns), the state is reset.

  - When the input size difference is in the length of channels (i.e., number of rows), there is no reset and the running operation is carried out as usual.

Resetting the Running Minimum

The block resets the running minimum whenever a reset event is detected at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

When a reset event occurs while the **Input processing** parameter is set to Elements as channels (sample based), the running minimum for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to Columns as channels (frame based), the running minimum for each channel is initialized to the earliest value in each channel of the current input.

You specify the reset event by the **Reset port** parameter:

- None — Disables the Rst port

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

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

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

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

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

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

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

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

Note When running simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

ROI Processing

To calculate the statistical value within a particular region of interest (ROI) of the input, select the **Enable ROI processing** check box. This applies to any mode other than the running mode and when you set the **Find the minimum value over** parameter to `Entire input` and you select the **Enable ROI processing** check box. ROI processing applies only for 2-D inputs.

Note Full ROI processing is only available to users who have a Computer Vision System Toolbox license. If you only have a DSP System Toolbox license, you can still use ROI processing, but are limited to the **ROI type** `Rectangles`.

Use the **ROI type** parameter to specify whether the ROI is a rectangle, line, label matrix, or binary mask. A binary mask is a binary image that enables you to specify which pixels to highlight, or select. In a label matrix, pixels equal to 0 represent the background, pixels equal to 1 represent the first object, pixels equal to 2 represent the second object, and so on. When the **ROI type** parameter is set to `Label matrix`, the Label and Label Numbers ports appear on the block. Use the Label Numbers port to specify the objects in the label matrix for which the block calculates statistics. The input to this port must be a vector of scalar values that correspond to the labeled regions in the label matrix. For more information about the format of the input to the ROI port when the ROI is a rectangle or a line, see the Draw Shapes block reference page.

For rectangular ROIs, use the **ROI portion to process** parameter to specify whether to calculate the statistical value for the entire ROI or just the ROI perimeter.

Use the **Output** parameter to specify the block output. The block can output separate statistical values for each ROI or the statistical value for all specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select `Binary mask`.

If, for the **ROI type** parameter, you select Rectangles or Lines, the **Output flag indicating if ROI is within image bounds** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

Output = Individual statistics for each ROI

Flag Port Output	Description
0	ROI is completely outside the input image.
1	ROI is completely or partially inside the input image.

Output = Single statistic for all ROIs

Flag Port Output	Description
0	All ROIs are completely outside the input image.
1	At least one ROI is completely or partially inside the input image.

If the ROI is partially outside the image, the block only computes the statistical values for the portion of the ROI that is within the image.

If, for the **ROI type** parameter, you select Label matrix, the **Output flag indicating if input label numbers are valid** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

Minimum

Output = Individual statistics for each ROI

Flag Port Output	Description
0	Label number is not in the label matrix.
1	Label number is in the label matrix.

Output = Single statistic for all ROIs

Flag Port Output	Description
0	None of the label numbers are in the label matrix.
1	At least one of the label numbers is in the label matrix.

Fixed-Point Data Types

The parameters on the **Data Types** pane of the block dialog are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 2-1060. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Examples   In the following ex_minimum_ref model, the Minimum block calculates the running minimum of a 3-by-2 matrix input. The **Input processing** parameter is set to Columns as channels (frame based), so the block processes the input as a two channel signal with a frame size of three. The running minimum is reset at *t*=2 by an impulse to the block's Rst port.
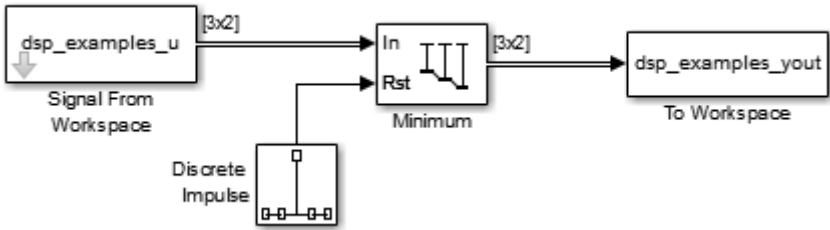
The operation of the block is shown in the following figure.
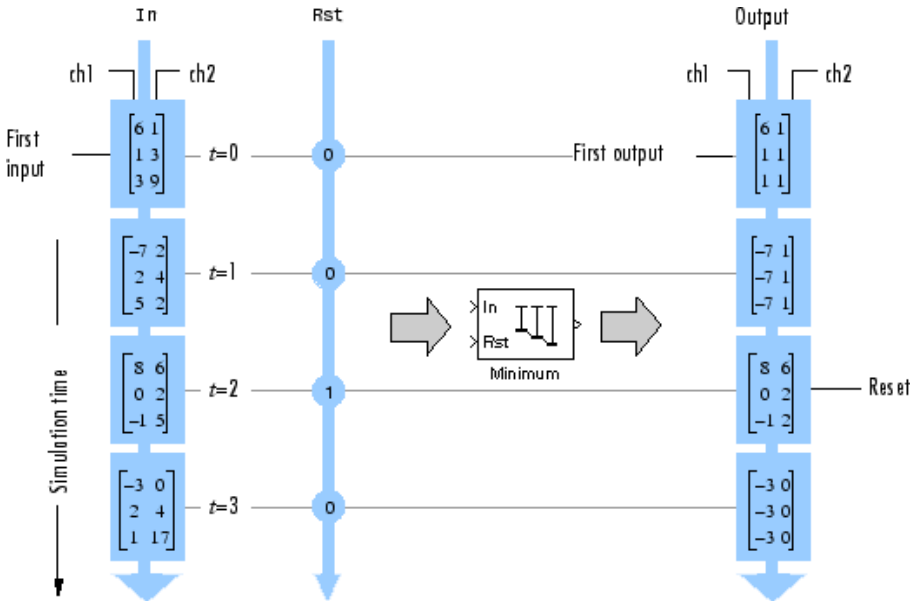
Minimum

The **Main** pane of the Minimum block dialog appears as follows.
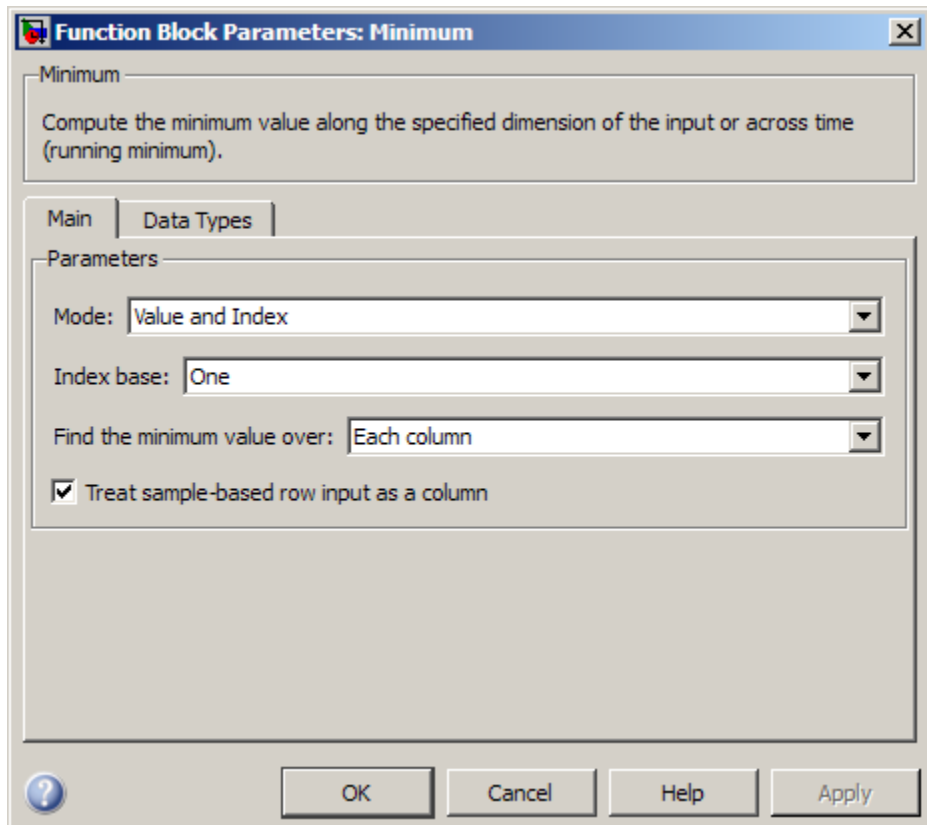
Mode

Specify the block's mode of operation:

- `Value` — Output the minimum value of each input

- `Index` — Output the index of the minimum value

- `Value and index` — Output both the value and the index

• `Running` — Track the minimum value of the input sequence over time

For more information, see Description.

Index base
Specify whether the index of the minimum value is reported using one-based or zero-based numbering. This parameter is only visible when the **Mode** parameter is set to `Index` or `Value and index`.

Find the minimum value over
Specify whether to find the minimum value along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see Description.

Input processing
Specify how the block should process the input when computing the running minimum. You can set this parameter to one of the following options:

• `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

• `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

This parameter appears only when you set the **Mode** to `Running`.

Note The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Minimum

Reset port
> Specify the reset event that causes the block to reset the running minimum. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you set the **Mode** parameter to Running. For information about the possible values of this parameter, see "Resetting the Running Minimum" on page 2-1064.

Treat sample-based row input as a column
> Select to treat sample-based length-*M* row vector inputs as *M*-by-1 column vectors. This parameter is only visible when the **Find the minimum value of** parameter is set to Each column.

> ---
> **Note** This check box will be removed in a future release. See "Sample-Based Row Vector Processing Changes" in the DSP System Toolbox Release Notes.
> ---

Dimension
> Specify the dimension (one-based value) of the input signal, over which the minimum is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the minimum value over** parameter is set to Specified dimension.

Enable ROI processing
> Select this check box to calculate the statistical value within a particular region of each image. This parameter is only available when the **Find the minimum value over** parameter is set to Entire input, and the block is not in running mode.

> ---
> **Note** Full ROI processing is only available to users who have a Computer Vision System Toolbox license. If you only have a DSP System Toolbox license, you can still use ROI processing, but are limited to the **ROI type** Rectangles.
> ---

The ROI processing parameters appear as follows.

ROI type

Specify the type of ROI you want to use. Your choices are
Rectangles, Lines, Label matrix, or Binary mask.

Minimum

ROI portion to process

Specify whether you want to calculate the statistical value for the entire ROI or just the ROI perimeter. This parameter is only visible if, for the **ROI type** parameter, you specify `Rectangles`.

Output

Specify the block output. The block can output a vector of separate statistical values for each ROI or a scalar value that represents the statistical value for all the specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select `Binary mask`.

Output flag

☑ **Output flag indicating if ROI is within image bounds**

☑ **Output flag indicating if label numbers are valid**

When you select either of these check boxes, the Flag port appears on the block. For a description of the Flag port output, see the tables in "ROI Processing" on page 2-1066.

The **Output flag indicating if ROI is within image bounds** check box is only visible when you select `Rectangles` or `Lines` as the **ROI type**.

The **Output flag indicating if label numbers are valid** check box is only visible when you select `Label matrix` for the **ROI type** parameter.

The **Data Types** pane of the Minimum block dialog appears as follows.

Note The parameters on the **Data Types** pane are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 2-1060. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Minimum

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-1068 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ![>>] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-1068 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ![>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Reset	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Idx	• Double-precision floating point • 32-bit unsigned integers
Val	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Minimum

See Also

Maximum	DSP System Toolbox
Mean	DSP System Toolbox
MinMax	Simulink
Histogram	DSP System Toolbox
min	MATLAB

Purpose   Compute estimate of autoregressive (AR) model parameters using modified covariance method

Library   Estimation / Parametric Estimation

dspparest3
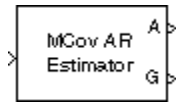
Description   The Modified Covariance AR Estimator block uses the modified covariance method to fit an autoregressive (AR) model to the input data. This method minimizes the forward and backward prediction errors in the least squares sense. The input is a frame of consecutive time samples, which is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, *A(z)*, independently for each successive input.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

You specify the order, p, of the all-pole model in the **Estimation order** parameter. To guarantee a valid output, you must set the **Estimation order** parameter to be less than or equal to two thirds the input vector length.

The output port labeled A outputs the normalized estimate of the AR model coefficients in descending powers of *z*.

[1 a(2) ... a(p+1)]

The scalar gain, *G*, is output from the output port labeled G.

See the Burg AR Estimator block reference page for a comparison of the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

Modified Covariance AR Estimator
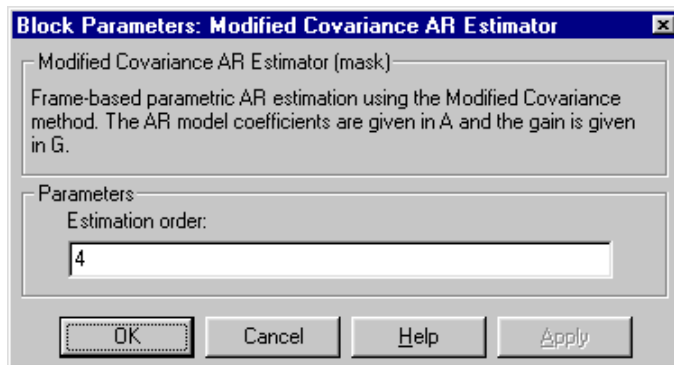
Estimation order

Specify the order of the AR model, *p*.

References
Kay, S. M. *Modern Spectral Estimation: Theory and Application.*
Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.*
Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
A	• Double-precision floating point
	• Single-precision floating point
G	• Double-precision floating point
	• Single-precision floating point

The output data type is the same as the input data type.

See Also

Burg AR Estimator	DSP System Toolbox
Covariance AR Estimator	DSP System Toolbox
Modified Covariance Method	DSP System Toolbox
Yule-Walker AR Estimator	DSP System Toolbox
armcov	Signal Processing Toolbox

Modified Covariance Method

Purpose   Power spectral density estimate using modified covariance method

Library   Estimation / Power Spectrum Estimation

dspspect3

Description
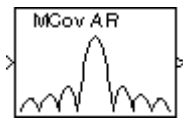
The Modified Covariance Method block estimates the power spectral density (PSD) of the input using the modified covariance method. This method fits an autoregressive (AR) model to the signal. It does so by minimizing the forward and backward prediction errors in the least squares sense. The **Estimation order** parameter value specifies the order of the all-pole model. To guarantee a valid output, the **Estimation order** parameter must be less than or equal to two thirds of the input vector length. The block computes the spectrum from the FFT of the estimated AR model parameters.

The input must be a sample-based vector (row, column, or 1-D) or frame-based vector (column only). This input represents a frame of consecutive time samples from a single-channel signal. The block outputs a column vector containing the estimate of the power spectral density of the signal at N_{fft} equally spaced frequency points. The frequency points are in the range $[0, F_s)$, where F_s is the sampling frequency of the signal.

Selecting **Inherit FFT length from estimation order**, specifies that N_{fft} is one greater than the estimation order. Clearing the **Inherit FFT length from estimation order** check box allows you to use the **FFT length** parameter to specify N_{fft} as a power of 2. The block zero-pads or wraps the input to N_{fft} before computing the FFT. The output is always sample based.

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must 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 equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

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 example compares the modified covariance method with several other spectral estimation methods.

Modified Covariance Method
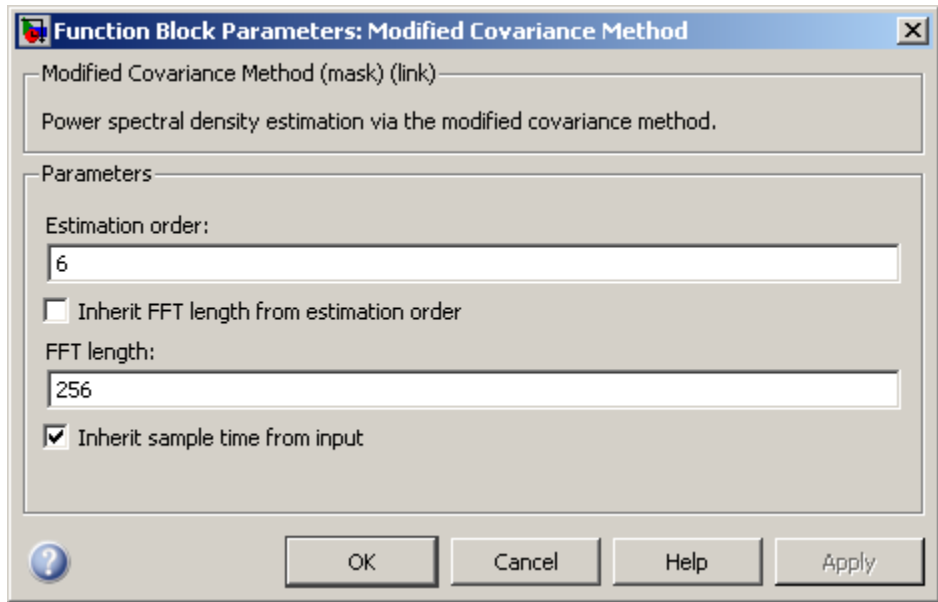
Estimation order

Specify the order of the AR model. To guarantee a valid output, the **Estimation order** parameter must be less than or equal to two thirds of the input vector length.

Inherit FFT length from estimation order

When you select this check box, the option specifies that the FFT length is one greater than the estimation order. To specify the number of points on which to perform the FFT, clear this check box. You can then specify a power of two FFT length using the **FFT length** parameter.

FFT length

Enter the number of data points, N_{fft}, on which to perform the FFT. When N_{fft} is larger than the input frame size, the block zero-pads each frame as needed. When N_{fft} is smaller than the input frame size, the block wraps each frame as needed. This

parameter becomes visible only when you clear the **Inherit FFT length from estimation order** check box.

Inherit sample time from input

If you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must 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 equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

Sample time of original time series

Specify the sample time of the original time-domain signal. This parameter becomes visible only when you clear the **Inherit sample time from input** check box.

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.

Orfanidis, S. J. *Introduction to Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1995.

Modified Covariance Method

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type.

See Also

Burg Method	DSP System Toolbox
Covariance Method	DSP System Toolbox
Modified Covariance AR Estimator	DSP System Toolbox
Short-Time FFT	DSP System Toolbox
Yule-Walker Method	DSP System Toolbox
spectrum.mcov	Signal Processing Toolbox

See "Spectral Analysis" for related information.

Purpose   Generate multiple binary clock signals

Library   • Sources

                     dspsrcs4

                     • Signal Management / Switches and Counters

                     dspswit3
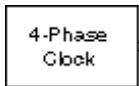
Description   The Multiphase Clock block generates a 1-by-*N* vector of clock signals,
                     where you specify the integer *N* in the **Number of phases** parameter.
                     Each of the *N* phases has the same frequency, *f*, specified in hertz by
                     the **Clock frequency** parameter.

```
4-Phase
Clock
```

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 output is therefore
$1/(N*f)$ seconds.

The active level can be either high (1) or low (0), as specified by the
Active level (polarity) parameter. The duration of the active level,
D, is set by the **Number of phase intervals over which the clock
is active**. This value, which can be an integer value between 1 and
N-1, specifies the number of phase intervals that each signal should
remain in the active state after becoming active. The active duty cycle
of the signal is *D/N*.

Examples   In the following ex_multiphaseclock_ref model, the Multiphase Clock
                    block generates a 100 Hz five-phase output in which the third signal is
                    first to become active. The block uses a high active level with a duration
                    of one interval.

Multiphase Clock
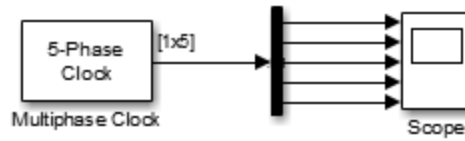
The Scope window below shows the Multiphase Clock block's output. Note that the first active level appears at *t*=0 on y(3), the second active level appears at *t*=0.002 on y(4), the third active level appears at *t*=0.004 on y(5), the fourth active level appears at *t*=0.006 on y(1), and the fifth active level appears at *t*=0.008 on y(2). Each signal becomes active 1/(5*100) seconds after the previous signal.
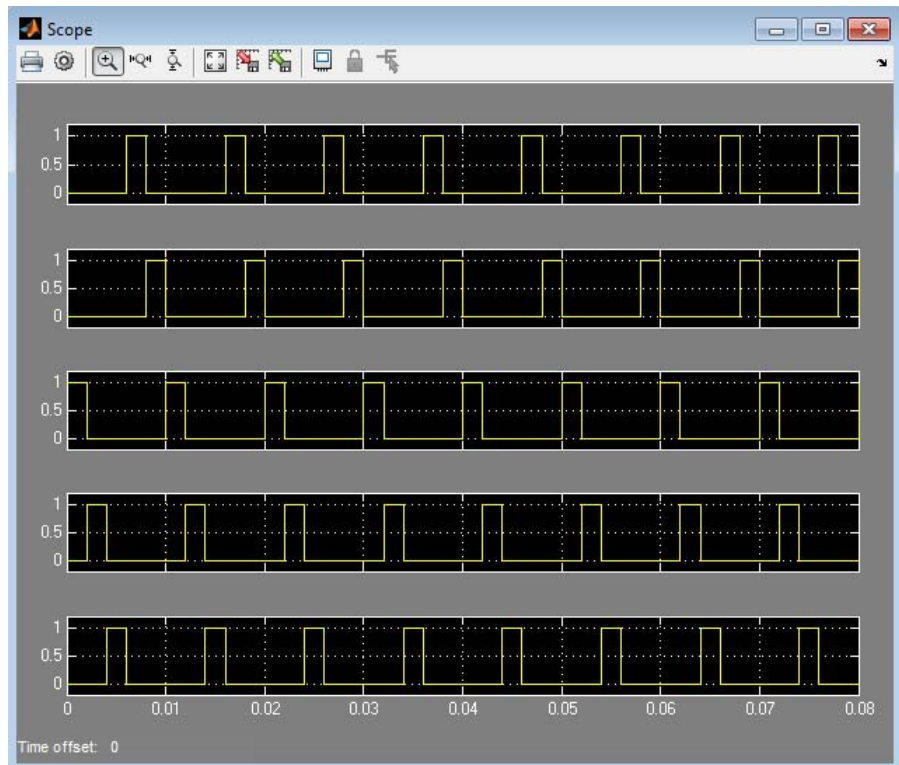
To experiment further, try changing the **Number of phase intervals over which clock is active** setting to 3 so that the active-level duration is three phase intervals (60% duty cycle).

Multiphase Clock
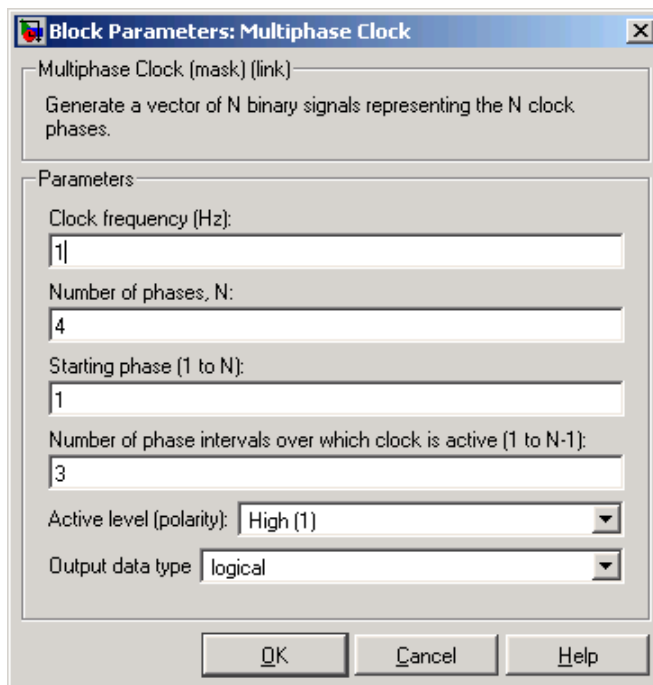
Clock frequency
    The frequency of all output clock signals.

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.

Number of phase intervals over which clock is active
    The duration of the active level for every output signal.

Active level
    The active level, High (1) or Low (0).

Output data type
    The output data type.

Supported Data Types

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

See Also

Clock	Simulink
Counter	DSP System Toolbox
Pulse Generator	Simulink
Event-Count Comparator	DSP System Toolbox

Multiport Selector

Purpose

Distribute arbitrary subsets of input rows or columns to multiple output ports

Library

Signal Management / Indexing

dspindex

Description

The Multiport Selector block extracts multiple subsets of rows or columns from *M*-by-*N* input matrix *u*, and propagates each new submatrix to a distinct output port. The block treats an unoriented length-*M* vector input 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 you set the **Select** parameter to Rows, the block uses the one-dimensional indices you specify to select matrix rows, and all elements on the chosen rows are included. When you set the **Select** parameter to Columns, the block uses the one-dimensional indices you specify to select matrix columns, and all elements on the chosen columns are included. A given input row or column can appear any number of times in any of the outputs, or not at all.

When an index references a nonexistent row or column of the input, the block reacts with the action you specify using the **Invalid index** parameter.

Examples

Example 1

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*

Multiport Selector
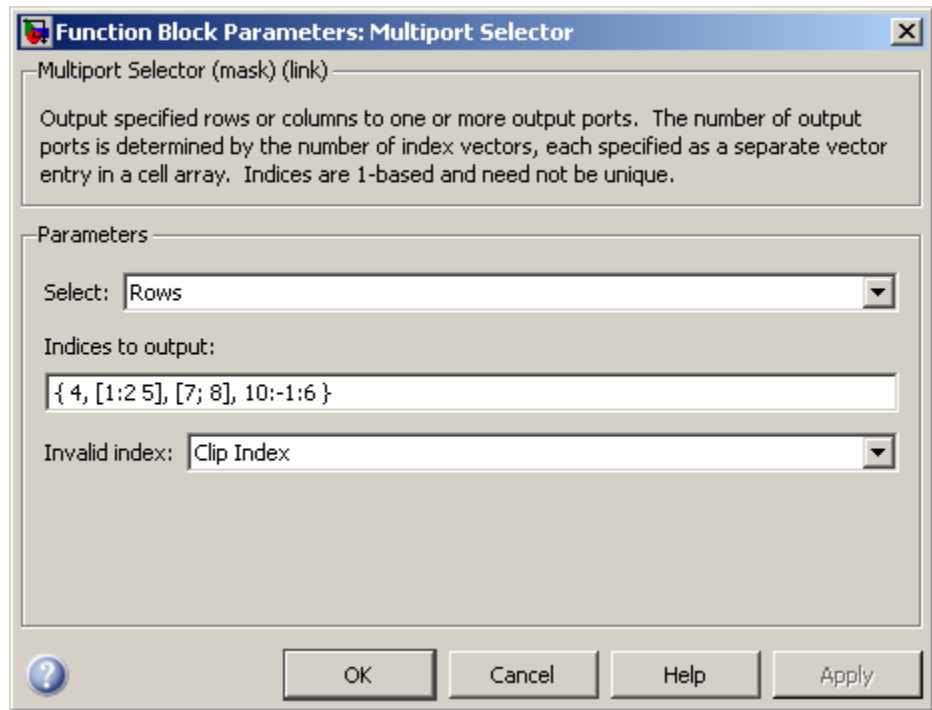
Dialog Box

Select
    Specify the dimension of the input to select, Rows or Columns.

Indices to output
    A cell array specifying the row- or column-subsets to propagate to each of the output ports. The number of cells in the array determines the number of output ports on the block.

Invalid index
    Specify how the block handles an invalid index value. You can select one of the following options:

    • Clip index — Clip the index to the nearest valid value, and do not issue an alert.

For example, if the block receives a 64-by-4 input and the **Select** parameter is set to Rows, the block clips an index of 72 to 64. For the same input, if the **Select** parameter is set to Columns, the block clips an index of 72 to 4. In both cases, the block clips an index of -2 to 1.

- Clip and warn — Clip the index to the nearest valid value and display a warning message at the MATLAB command line.

- Generate error — Display an error dialog box and terminate the simulation.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	• Enumerated
Outputs	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	• Enumerated

Multiport Selector

See Also

Permute Matrix	DSP System Toolbox
Selector	Simulink
Submatrix	DSP System Toolbox
Variable Selector	DSP System Toolbox

Purpose   Output ones or zeros for specified number of sample times

Library   • Sources

dspsrcs4

• Signal Management / Switches and Counters

dspswit3
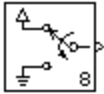
Description   The N-Sample Enable block outputs the inactive value (0 or 1, whichever is not selected in the **Active level** parameter) during the first N sample times, where N is the **Trigger count** value. Beginning with output sample N+1, the block outputs the active value (1 or 0, whichever you select in the **Active level** parameter) until a reset event occurs or the simulation terminates.

The output of the block is always a scalar.

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. This block supports triggered subsystems when you select the **Reset input** check box.

You specify the triggering event in the **Trigger type** pop-up menu:

• Rising edge — Triggers a reset operation when the Rst input does one of the following:

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

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

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

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

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

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

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

Dialog Box

Trigger count
> Specify the number of samples for which the block outputs the active value. Tunable.

Active level
> Specify the value to output after the first *N* sample times, 0 or 1. Tunable.

Reset input
> Select to enable the Rst input port.

N-Sample Enable

Trigger type

Select type of event that triggers a reset when the Rst port is enabled.

Sample time

Specify the sample period, T_s, for the block's counter. The block switches from the active value to the inactive value at $t=T_s*(N+1)$.

Output data type

Select the output data type.

Supported Data Types

- Double-precision floating point
- Boolean — The block accepts Boolean inputs to the Rst port, which is enabled when you select the **Reset input** parameter.

See Also

Counter	DSP System Toolbox
N-Sample Switch	DSP System Toolbox

Purpose   Switch between two inputs after specified number of sample periods

Library   Signal Management / Switches and Counters

dspswit3
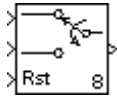
Description   The N-Sample Switch block outputs the signal connected to the top
input port during the first *N* sample times after the simulation begins
or the block is reset, where you specify *N* in the **Switch count**
parameter. Beginning with output sample *N*+1, the block outputs the
signal connected to the bottom input until the next reset event or the
end of the simulation.

You specify the sample period of the output in the **Sample time**
parameter (that is, the output sample period is not inherited from the
sample period of either input). The block applies a zero-order hold at
the input ports, so the value the block reads from a given port between
input sample times is the value of the most recent input to that port.

Both inputs must have the same dimension, except in the following
two cases:

• When one input is a scalar, the block expands the scalar input to
  match the size of the other input.

• When one input is an unoriented vector and the other input is a
  row or column vector with the same number of elements, the block
  reshapes the unoriented vector to match the dimension of the other
  input.
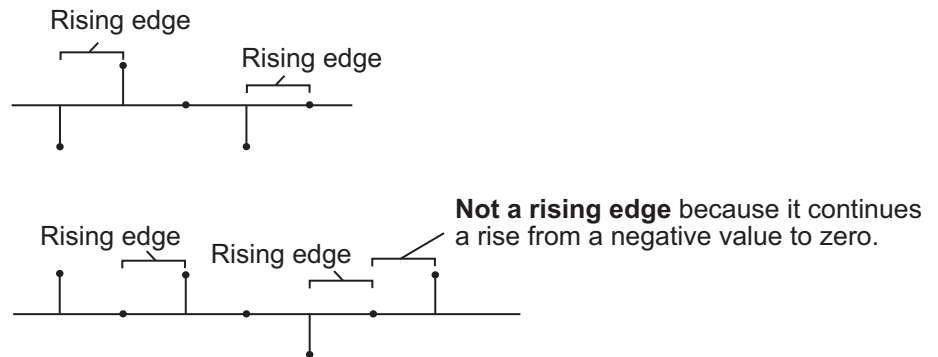
The **Reset input** check box enables the Rst input port. At any time
during the count, a trigger event at the Rst port resets the counter to
zero. The reset sample time must be a positive integer multiple of the
input sample time. This block supports triggered subsystems when you
select the **Reset input** check box.

You specify the triggering event in the **Trigger type** pop-up menu,
and can be one of the following:

N-Sample Switch

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

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

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

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

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

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

Falling edge   Falling edge

Falling edge   **Not a falling edge** because it continues
a fall from a positive value to zero.

Falling edge

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

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

N-Sample Switch
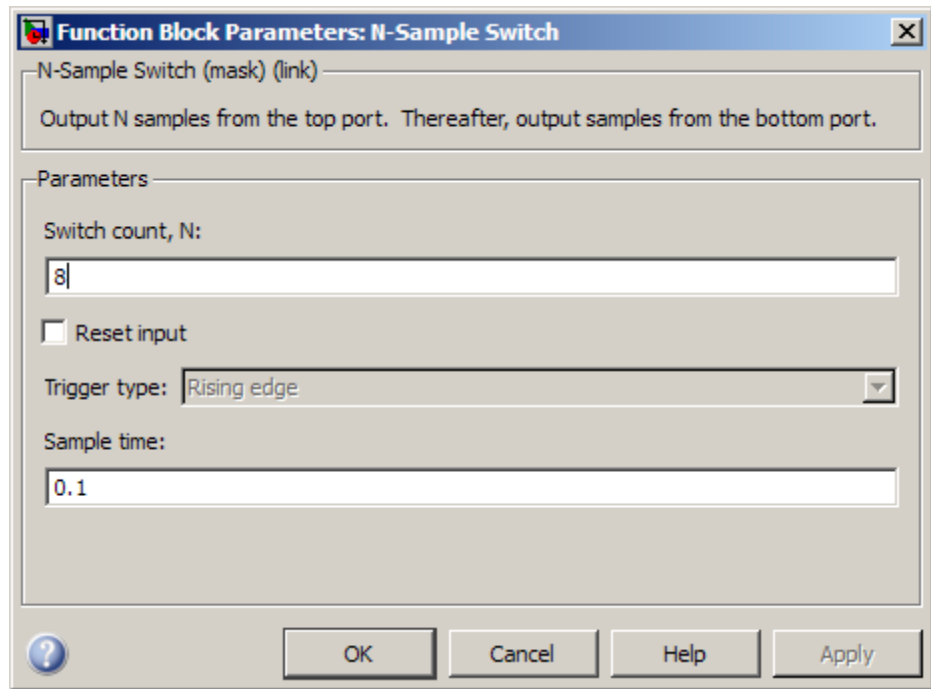
Switch count

The number of sample periods, *N*, for which the output is connected to the top input before switching to the bottom input. Tunable.

Reset input

Enables the Rst input port when selected. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

Trigger type

The type of event at the Rst port that resets the block's counter. This parameter is enabled when you select **Reset input**. Tunable.

Sample time

The sample period, T_s, for the block's counter. The block switches inputs at $t=T_s*(N+1)$.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed and unsigned)

- Boolean — The block accepts Boolean inputs to the Rst port, which is enabled when you set the **Reset input** parameter.

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

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

See Also

Counter	DSP System Toolbox
N-Sample Enable	DSP System Toolbox

NCO

Purpose   Generate real or complex sinusoidal signals

Library   Signal Operations

dspsigops
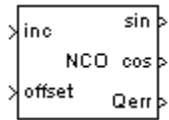
Description   The NCO block generates a multichannel real or complex sinusoidal signal, with independent frequency and phase in each output channel. The amplitude of the created signal is always 1. The block implements the algorithm as shown in the following diagram:
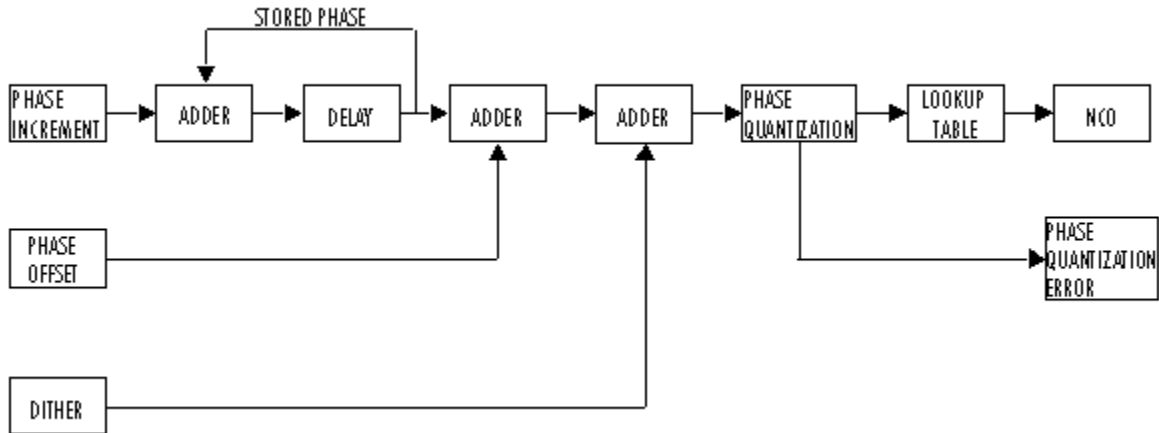
The implementation of a numerically controlled oscillator (NCO) has two distinct parts. First, a phase accumulator accumulates the phase increment and adds in the phase offset. In this stage, an optional internal dither signal can also be added. The NCO output is then calculated by quantizing the results of the phase accumulator section and using them to select values from a lookup table.

Given a desired output frequency F_0, calculate the value of the **Phase increment** block parameter with

$$phase\ increment = (\frac{F_0 \cdot 2^N}{F_s})$$

where N is the accumulator word length and

$$F_s = \frac{1}{T_s} = \frac{1}{sample\ time}$$

The frequency resolution of an NCO is defined by

$$\Delta f = \frac{1}{T_s \cdot 2^N}\ \text{Hz}$$

Given a desired phase offset (in radians), calculate the **Phase offset** block parameter with

$$phase\ offset = \frac{2^N \cdot desired\ phase\ offset}{2\pi}$$

The spurious free dynamic range (SFDR) is estimated as follows for a lookup table with 2^P entries, where P is the number of quantized accumulator bits:

$$SFDR = (6P)\ \text{dB} \qquad \text{without dither}$$
$$SFDR = (6P + 12)\ \text{dB} \quad \text{with dither}$$

This block uses a quarter-wave lookup table technique that stores table values from 0 to $\pi/2$. The block calculates other values on demand using the accumulator data type, then casts them into the output data type. This can lead to quantization effects at the range limits of a given data type. For example, consider a case where you would expect the value of the sine wave to be –1 at π. Because the lookup table value at that point must be calculated, the block might not yield exactly –1, depending on the precision of the accumulator and output data types.

NCO

The NCO block supports real inputs only. All outputs are real except for the output signal in `Complex exponential` mode.

To produce a multichannel output, specify a vector quantity for the **Phase increment** and **Phase offset** parameters. Both parameters must have the same length, which defines the number of output channels. Each element of each vector is applied to a different output channel.

Fixed-Point Data Types

The following diagram shows the data types used within the NCO block.

I = Integer
A = Accumulator data type
D = Dither bits
Q = Quantized accumulator bits
P = Phase quantization data type
O = Output data type
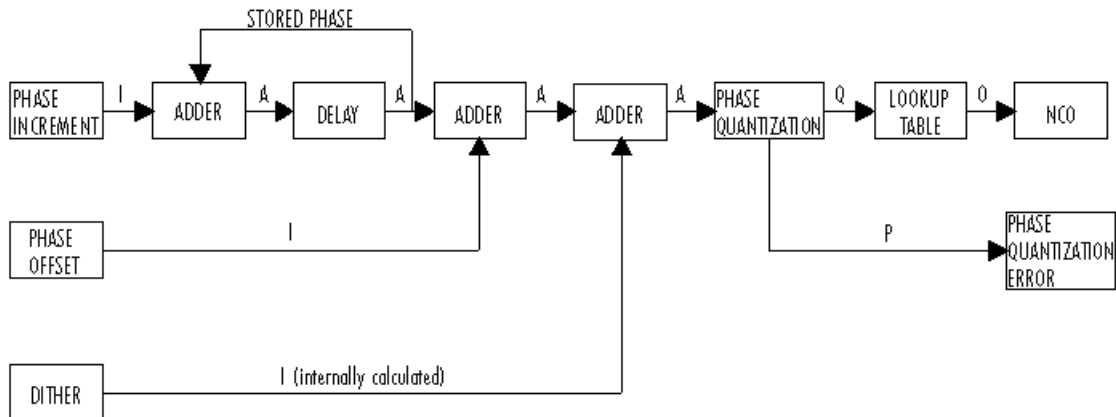
- You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 2-1113 below.

> **Note** The lookup table for this block is constructed from double-precision floating-point values. Thus, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** data type to values greater than 53 bits does not improve the precision of your output.

- The phase increment and phase offset inputs must be integers or fixed-point data types with zero fraction length.

- You specify the number of quantized accumulator bits in the **Number of quantized accumulator bits** parameter.

- The phase quantization error word length is equal to the accumulator word length minus the number of quantized accumulator bits, and the fraction length is zero.

Examples   The NCO block is used in the GSM Digital Down Converter product example. Open this example by typing dspddc at the MATLAB command line.

You can also try the following example. Design an NCO source with the following specifications:

- Desired output frequency $F_0 = 510$ Hz

- Frequency resolution $\Delta f = 0.05$ Hz

- Spurious free dynamic range $SFDR \geq 90$ dB

- Sample period $T_s = 1/8000$ s

- Desired phase offset $\pi/2$

1

Calculate the number of required accumulator bits from the equation for frequency resolution:

$$\Delta f = \frac{1}{T_s \cdot 2^N} \, \text{Hz}$$

$$0.05 = \frac{1}{\frac{1}{8000} \cdot 2^N} \, \text{Hz}$$

$$N = 18$$

Note that N must be an integer value. The value of N is rounded up to the nearest integer; 18 accumulator bits are needed to accommodate the value of the frequency resolution.

2

Using this best value of N, calculate the frequency resolution that will be achieved by the NCO block:

$$\Delta f = \frac{1}{T_s \cdot 2^N} \, \text{Hz}$$

$$\Delta f = \frac{1}{\frac{1}{8000} \cdot 2^{18}} \, \text{Hz}$$

$$\Delta f = 0.0305$$

3

Calculate the number of quantized accumulator bits from the equation for spurious free dynamic range and the fact that for a lookup table with 2^P entries, P is the number of quantized accumulator bits:

$$SFDR = (6P + 12) \, \text{dB}$$

$$96 \, \text{dB} = (6P + 12) \, \text{dB}$$

$$P = 14$$

4

Select the number of dither bits. In general, a good choice for the number of dither bits is the accumulator word length minus the number of quantized accumulator bits; in this case 4.

5

Calculate the phase increment:

$$phase\ increment = \text{round}(\frac{F_0 \cdot 2^N}{F_s})$$

$$phase\ increment = \text{round}(\frac{510 \cdot 2^{18}}{8000})$$

$$phase\ increment = 16712$$

6

Calculate the phase offset:

$$phase\ offset = \frac{2^{accumulator\ word\ length} \cdot desired\ phase\ offset}{2\pi}$$

$$phase\ offset = \frac{2^{18} \cdot \frac{\pi}{2}}{2\pi}$$

$$phase\ offset = 65536$$

7

Type `ex_nco_example` at the MATLAB command line to open the following model:
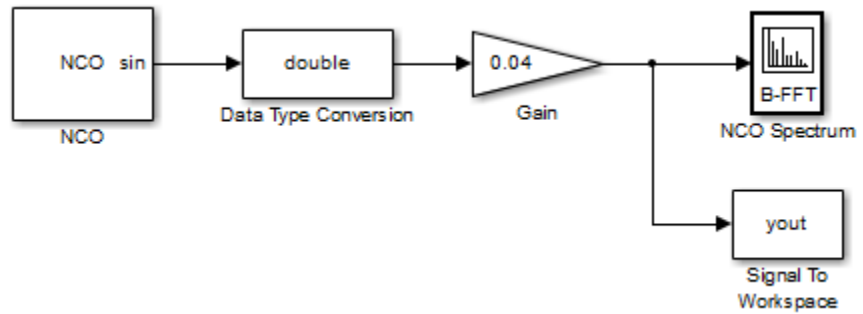
The NCO block in the model is populated with the specifications and quantities you just calculated. The output word length and fraction length depend on the constraints of your hardware; this example uses a word length of 16 and a fraction length of 14. You can verify that the specifications of this problem have been met by looking at the **NCO Characterization** pane of the NCO block.

8

Experiment with the model to observe the effects on the output shown on the Spectrum Scope. For example, try turning dithering on and off, and try changing the number of dither bits.

Dialog Box

The **Main** pane of the NCO dialog appears as follows.
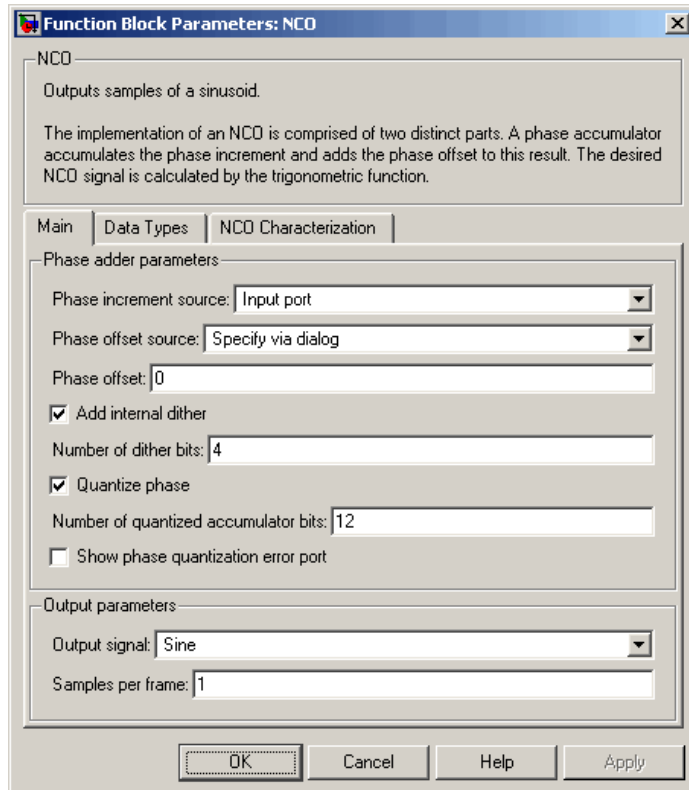
Phase increment source

Choose how you specify the phase increment. The phase increment can come from an input port or from the dialog.

- If you select Input port, the inc port appears on the block icon.

- If you select `Specify via dialog`, the **Phase increment** parameter appears.

Phase increment

Specify the phase increment. Only integer data types, including fixed-point data types with zero fraction length, are allowed. The dimensions of the phase increment are dictated by those of the phase offset:

- When you specify the phase offset on the block dialog box, the phase increment must be a scalar or a vector with the same length as the phase offset. The block applies each element of the vector to a different channel, and therefore the vector length defines the number of output channels.

- When you specify the phase offset via an input port, the offset port treats each column of the input as an independent channel. The phase increment length must equal the number of columns in the input to the offset port.

This parameter is visible only if you set the **Phase increment source** parameter to `Specify via dialog`.

Phase offset source

Choose how you specify the phase offset. The phase offset can come from an input port or from the dialog.

- If you select `Input port`, the offset port appears on the block icon.

- If you select `Specify via dialog`, the **Phase offset** parameter appears.

When you specify the phase offset via an input port, it can be a scalar, vector, or a full matrix. The block treats each column of the input to the offset port as an independent channel. The number of channels in the phase offset must match the number of channels in the data input. For each frame of the input, the block can apply different phase offsets to each sample and channel.

Only integer data types, including fixed-point data types with zero fraction length, are allowed.

Phase offset

Specify the phase offset. When you specify the phase offset using this parameter rather than via an input port, it must be a scalar or vector with the same length as the phase increment. Scalars are expanded to a vector with the same length as the phase increment. Each element of the phase offset vector is applied to a different channel of the input, and therefore the vector length defines the number of output channels. Only integer data types, including fixed-point data types with zero fraction length, are allowed.

This parameter is visible only if `Specify via dialog` is selected for the **Phase offset source** parameter.

Add internal dither

Select to add internal dithering to the NCO algorithm. Dithering is added using the PN Sequence Generator from the Communications System Toolbox™ product.

Number of dither bits

Specify the number of dither bits.

This parameter is visible only if **Add internal dither** is selected.

Quantize phase

Select to enable quantization of the accumulated phase.

Number of quantized accumulator bits

Specify the number of quantized accumulator bits. This determines the number of entries in the lookup table. The number of quantized accumulator bits must be less than the accumulator word length.

This parameter is visible only if **Quantize phase** is selected.

Show phase quantization error port

Select to output the phase quantization error. When you select this, the Qerr port appears on the block icon.

This parameter is visible only if **Quantize phase** is selected.

Output signal

Choose whether the block should output a `Sine`, `Cosine`, `Complex exponential`, or `Sine and cosine` signals. If you select `Sine and cosine`, the two signals output on different ports.

Sample time

Specify the sample time in seconds when the block is acting as a source. When either the phase increment or phase offset come in via block input ports, the sample time is inherited and this parameter is not visible.

Samples per frame

Specify the number of samples per frame. When the value of this parameter is 1, the block outputs a sample-based signal. When the value is greater than 1, the block outputs a frame-based signal of the specified size. In frame-based mode, the phase increment and phase offset can vary from channel to channel and from frame to frame, but they are constant along each channel in a given frame.

When the phase offset input port exists, it has the same frame status as any output port present. When the phase increment input port exists, it does not support frames.

This parameter is only visible if either **Phase increment source** or **Phase offset source** is set to `Specify via dialog`.

The **Data Types** pane of the NCO dialog appears as follows.
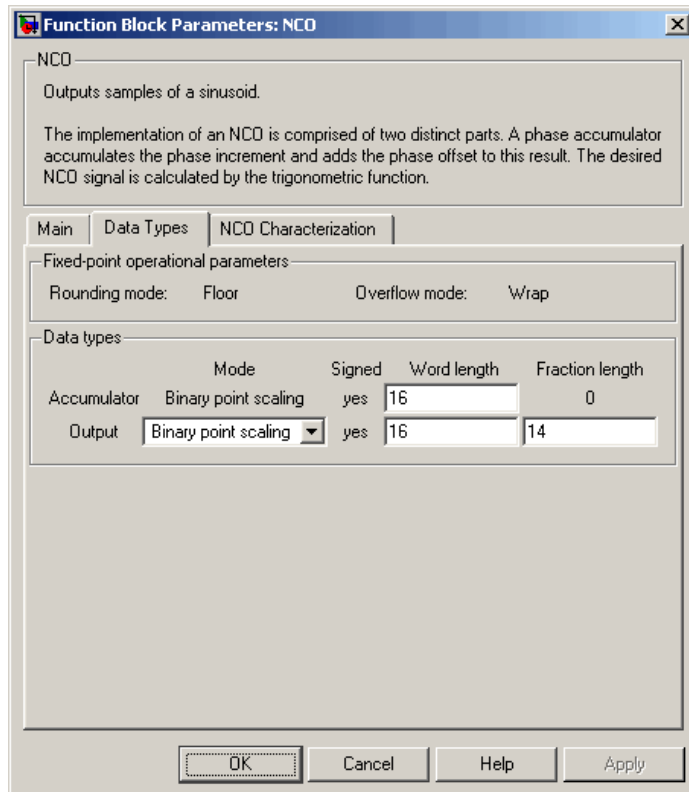
Rounding mode

The rounding mode used for this block when inputs are fixed point is always Floor.

Overflow mode

The overflow mode used for this block when inputs are fixed point is always Wrap.

Accumulator

Specify the word length of the accumulator data type. The fraction length is always zero; this is an integer data type.

Output

Specify the output data type.

- Choose double or single for a floating-point implementation.

- When you select Binary point scaling, you can enter the word length and the fraction length of the output, in bits.

Note The lookup table for this block is constructed from double-precision floating-point values. Thus, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** data type to values greater than 53 bits does not improve the precision of your output.

The **NCO Characterization** pane of the NCO dialog appears as follows.
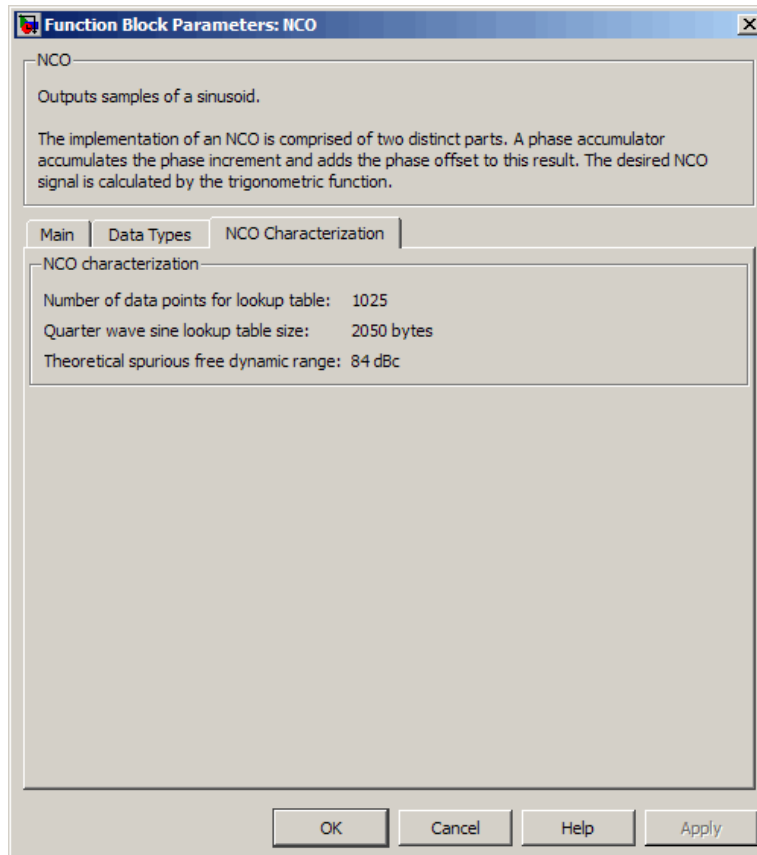
The **NCO Characterization** pane does not have any parameters. Instead, it provides you with details on the NCO signal currently being implemented by the block:

- `Number of data points for lookup table` — The lookup table is implemented as a quarter-wave sine table. The number of lookup table data points is defined by

$$2^{\text{number of quantized accumulator bits}-2} + 1$$

- Quarter wave sine lookup table size — The quarter wave sine lookup table size is defined by

$$\frac{(\text{number of data points for lookup table}) \cdot (\text{output word length})}{8} \text{ bytes}$$

- Theoretical spurious free dynamic range — The spurious free dynamic range (SFDR) is calculated as follows for a lookup table with 2^P entries:

$$SFDR = (6P) \text{ dB} \qquad \text{without dither}$$
$$SFDR = (6P + 12) \text{ dB} \quad \text{with dither}$$

Supported Data Types

Port	Supported Data Types
inc	• Fixed point with zero fraction length
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
offset	• Fixed point with zero fraction length
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
sin	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
Qerr	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

PN Sequence Generator	Communications System Toolbox
Sine Wave	DSP System Toolbox

Normalization

Purpose   Perform vector normalization along rows, columns, or specified dimension

Library   Math Functions / Math Operations

dspmathops

Description

$$\frac{u}{\|u\|^2}$$
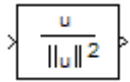
The Normalization block independently normalizes each row, column, or vector of the specified dimension of the input. The block accepts both fixed- and floating-point signals in the squared 2-norm mode, but only floating-point signals in the 2-norm mode. The output always has the same dimensions as the input.

This block treats an arbitrarily dimensioned input U as a collection of vectors oriented along the specified dimension. The block normalizes these vectors by either their norm or the square of their norm.

For example, consider a 3-dimensional input $U(i,j,k)$ and assume that you want to normalize along the second dimension. First, define the 2-dimensional intermediate quantity $V(i,k)$ such that each element of V is the norm of one of the vectors in U:

$$V(i,k) = \left(\sum_{j=1}^{J} U^2(i,j,k) \right)^{1/2}$$

Given V, the output of the block $Y(i, j,k)$ in 2-norm mode is

$$Y(i,j,k) = \frac{U(i,j,k)}{V(i,k) + b}$$

In squared 2-norm mode, the block output is

$$Y(i,j,k) = \frac{U(i,j,k)}{V(i,k)^2 + b}$$

The normalization bias, b, is typically chosen to be a small positive constant (for example, 1e-10) that prevents potential division by zero.

Fixed-Point Data Types

The following diagram shows the data types used within the Normalization block for fixed-point signals (squared 2-norm mode only).
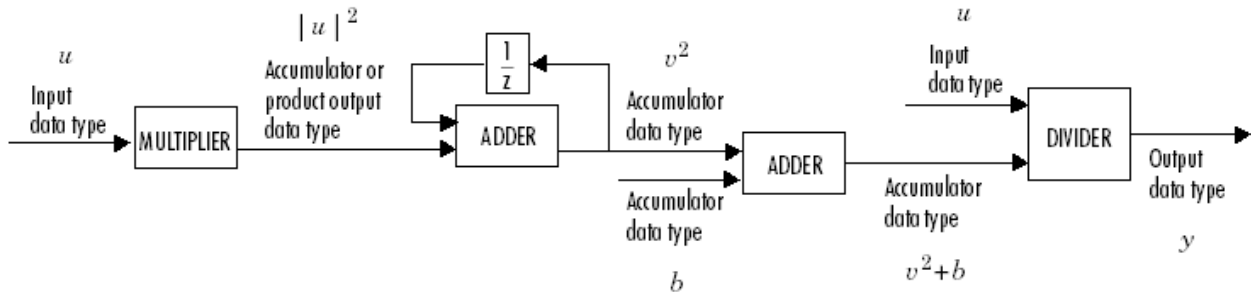
The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types". You can set the accumulator, product output, and output data types in the block dialog as discussed in "Dialog Box" on page 2-1123.

Examples   See "Zero Algorithmic Delay" in the *DSP System Toolbox User's Guide* for an example.

Dialog Box   The **Main** pane of the Normalization dialog appears as follows.
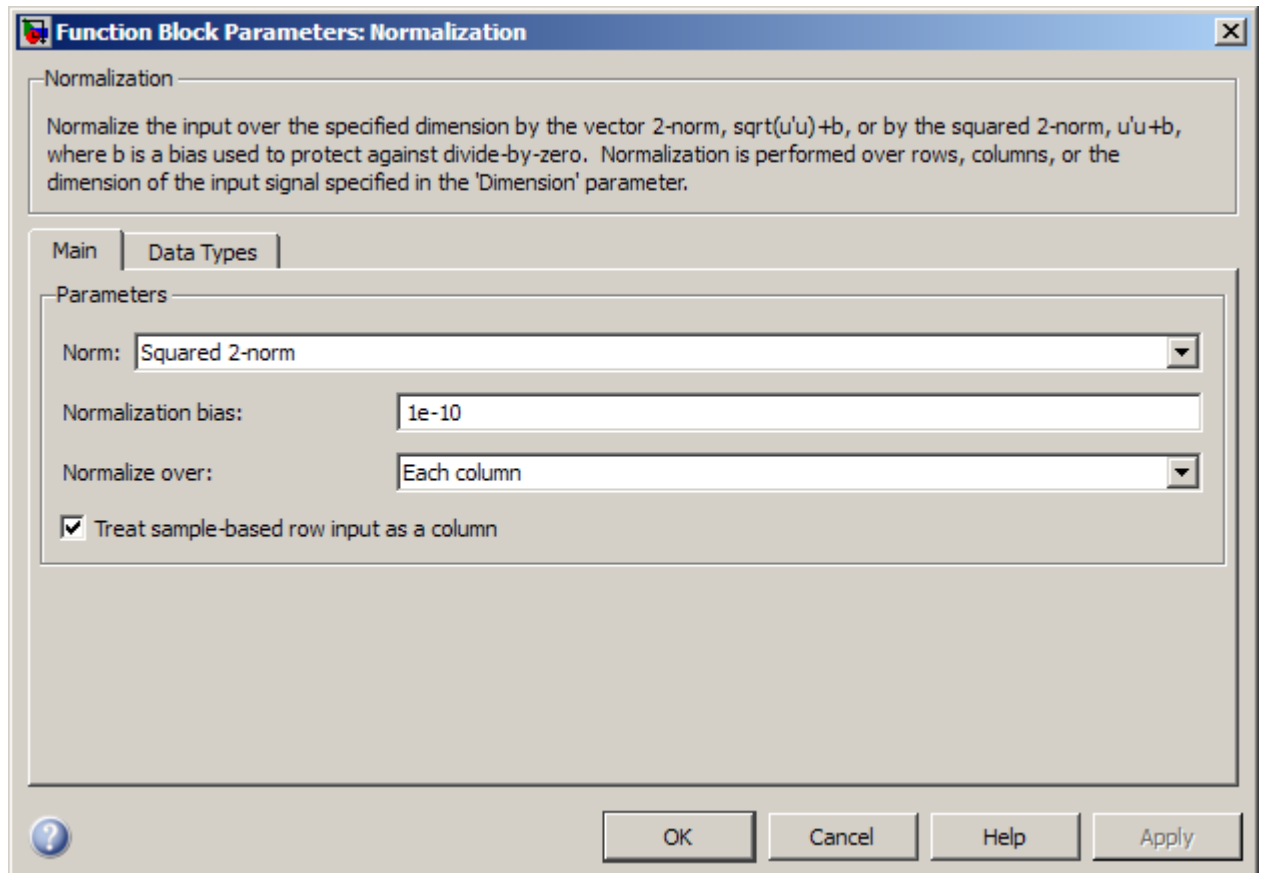
Normalization

Norm

    Specify the type of normalization to perform, `2-norm` or `Squared 2-norm`. `2-norm` mode supports floating-point signals only. `Squared 2-norm` supports both fixed-point and floating-point signals.

Normalization bias

    Specify the real value b to be added in the denominator to avoid division by zero. Tunable.

Normalize over

Specify whether to normalize along rows, columns, or the dimension specified in the **Dimension** parameter.

Dimension

Specify the one-based value of the dimension over which to normalize. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible if `Specified dimension` is selected for the **Normalize over** parameter.

Treat sample-based row input as a column

Select to treat a sample-based row input as a column.

> **Note** This check box will be removed in a future release. See "Sample-Based Row Vector Processing Changes" in the DSP System Toolbox Release Notes.

The **Data Types** pane of the Normalization dialog appears as follows.

Normalization

Function Block Parameters: Normalization ✕

Normalization

Normalize the input over the specified dimension by the vector 2-norm, sqrt(u'u)+b, or by the squared 2-norm, u'u+b, where b is a bias used to protect against divide-by-zero. Normalization is performed over rows, columns, or the dimension of the input signal specified in the 'Dimension' parameter.

Main	Data Types

Fixed-point operational parameters

Rounding mode: | Floor ▾ | Overflow mode: | Wrap ▾ |

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.

	Data Type	Assistant	Minimum	Maximum
Product output:	Inherit: Same as input ▾	>>		
Accumulator:	Inherit: Same as product output ▾	>>		
Output:	Inherit: Same as product output ▾	>>	[ ]	[ ]

☐ Lock data type settings against changes by the fixed-point tools

? | OK | Cancel | Help | Apply |

Note The parameters on this pane are only applicable to fixed-point signals when the block is in squared 2-norm mode. See "Fixed-Point Data Types" on page 2-1123 for a diagram of how the product output, accumulator, and output data types are used in this case.

Rounding mode

  Select the rounding mode for fixed-point operations.

Overflow mode

  Select the overflow mode for fixed-point operations.

Product output data type

  Specify the product output data type. See "Fixed-Point Data Types" on page 2-1123 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

  Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

  See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

  Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-1123 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

  Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type
Specify the output data type. See "Fixed-Point Data Types" on page 2-1123 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum
Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum
Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
> Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Array-Vector Multiply	DSP System Toolbox
Reciprocal Condition	DSP System Toolbox
norm	MATLAB

Nyquist Filter

Purpose　　Design Nyquist filter

Library　　Filtering / Filter Designs

　　　　　　　dspfdesign

Description　This block brings the filter design capabilities of the `filterbuilder` function to the Simulink environment.

```
Nyquist
```

Dialog Box　See "Nyquist Filter Design Dialog Box — Main Pane" on page 7-750 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.

Function Block Parameters: Nyquist Filter

Nyquist Filter

Design a Nyquist filter.

[View Filter Response]

Filter specifications

Band: `2`

Impulse response: `FIR` ▼

Filter order mode: `Minimum` ▼

Filter Type: `Single-rate` ▼

Frequency specifications

Frequency units: `Normalized (0 to 1)` ▼   Input Fs: `2`

Transition width: `.1`

Magnitude specifications

Magnitude units: `dB` ▼

Astop: `80`

Algorithm

Design method: `Kaiser window` ▼

Filter Implementation

Structure: `Direct-form FIR` ▼

☐ Use basic elements to enable filter customization

Input processing: `Columns as channels (frame based)` ▼

☐ Use symbolic names for coefficients

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

Nyquist Filter

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify your filter format, such as the impulse response and the filter order.

Band

Specifies the location of the center of the transition region between the passband and the stopband. The center of the transition region, F_c, is calculated using the value for **Band**:

$$F_c = F_s/(2 \cdot \textbf{Band}).$$

The default value, 2, corresponds to a halfband filter.

Impulse response

Select either `FIR` or `IIR` from the drop-down list. `FIR` is the default. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly. These options are both available only when **Band** is 2. For values of **Band** greater than 2, only FIR designs are supported.

> **Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Filter order mode

Select either `Minimum` (the default) or `Specify` from the drop-down list. Selecting `Specify` enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select `Single-rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, the block specifies a single-rate filter.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Filter order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default value is 2.

Frequency Specifications

The parameters in this group allow you to specify your filter response curve.

Nyquist Filter

Frequency constraints

Select the filter features that the block uses to define the frequency response characteristics.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude Specifications

Parameters in this group specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. From the drop-down list, select one of the following options:

- `Linear` — Specify the magnitude in linear units.

- `dB` — Specify the magnitude in decibels (default)

- `Squared` — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure of your filter.

Design Method

> Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is `Butterworth`, and the default FIR method is `Kaiser window`.

Scale SOS filter coefficients to reduce chance of overflow

> Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

> The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

> **Density factor**
>
> > Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).
> >
> > Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of

20 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select Any, Even, or Odd from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- Flat — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- Linear — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to Linear, enter the slope of the stopband in units of dB/rad/s. The block applies that slope to the stopband.

- When you set **Stopband shape** to 1/f, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. The block applies the $(1/f)^n$ relation to

the stopband to result in an exponentially decreasing stopband attenuation.

Filter Implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

> **Note** The `Inherited (this choice will be removed see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.

- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Nyquist Filter

value

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Purpose   Design octave filter

Library   Filtering / Filter Designs

dspfdesign

Description

```
   Octave
```

This block brings the filter design capabilities of the filterbuilder function to the Simulink environment.
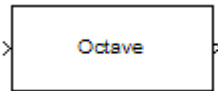
Dialog Box

See "Octave Filter Design Dialog Box — Main Pane" on page 7-758 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

Parameters of this block that do not change filter order or structure are tunable.

Octave Filter

Design an octave filter.

Function Block Parameters: Octave Filter

Octave Filter

Design an octave filter.

[View Filter Response]

Filter specifications

Order: 6

Bands per octave: 1

Frequency units: Hz   Input Fs: 48000

Center frequency: 1000

Algorithm

Design method: Butterworth

☑ Scale SOS filter coefficients to reduce chance of overflow

Filter Implementation

Structure: Direct-form II SOS

☐ Use basic elements to enable filter customization

☐ Optimize for unit scale values

Input processing: Columns as channels (frame based)

☐ Use symbolic names for coefficients

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

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

Order

Specify filter order. Possible values are: `4, 6, 8, 10`.

Bands per octave

Specify the number of bands per octave. Possible values are: `1, 3, 6, 12, 24`.

Frequency units

Specify frequency units as `Hz` or `kHz`.

Input Fs

Specify the input sampling frequency in the frequency units specified previously.

Center Frequency

Select from the drop-down list of available center frequency values.

Algorithm

Design Method

Butterworth is the design method used for this type of filter.

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Octave Filter

Filter Implementation

Structure

Specify filter structure. Choose from:

- Direct-form I SOS

- Direct-form II SOS

- Direct-form I transposed SOS

- Direct-form II transposed SOS

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Note The Inherited (this choice will be removed — see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- Enforce single-rate processing — When you select this option, the block maintains the sample rate of the input.

- Allow multirate processing — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to Elements as channels (sample based).

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows

tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

Purpose   Truncate vectors by removing or keeping beginning or ending values

Library   Signal Operations

dspsigops

Description
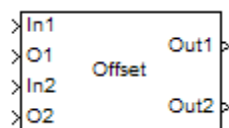
The Offset block removes or keeps values from the beginning or end of the input vectors. You specify the length of the output vectors using the **Output port length** parameter. The inputs to the In ports (In1, In2, ...) can be scalars or vectors, but they must be the same size and data type. The offset values are the inputs to the O ports (O1, O2, ...); they must be scalar values with the same data type. These offset values should be integer values because they determine the number of values the block discards or retains from each input vector. The block rounds any offset value that is a noninteger value to the nearest integer value. There is one output port for each pair of In and O ports.

Use the **Mode** parameter to determine which values the block discards or retains from the input vector. To discard the initial values of the vector, select Remove beginning samples. To discard the final values of the vector, select Remove ending samples. To retain the initial values of the vector, select Keep beginning samples. To retain the final values of a vector, select Keep ending samples.

Use the **Number of input data-offset pairs** parameter to specify the number of inputs to the block. The number of input ports is twice the scalar value you enter. For example, if you enter 3, ports In1, O1, In2, O2, In3, and O3 appear on the block.

The block uses the **Output port length** parameter to determine the length of the output vectors. If you select Same as input, the block outputs vectors that are the same length as the input to the In ports. If you select User-defined, the **Output length** parameter appears. Enter a scalar that represents the desired length of the output vectors. If your desired output length is greater than the number of values you extracted from your input vector, the block zero-pads the end of the vector to reach the length you specified.

Offset

Use the **Action for out of range offset value** parameter to determine how the block behaves when an offset value is not in the range $0 \leq$ offset value $\leq N$, where N is the input vector length. Select Clip if you want any offset values less than 0 to be set to 0 and any offset values greater than N to be set to N. Select Clip and warn if you want to be warned when any offset values less than 0 are set to 0 and any offset values greater than N are set to N. Select Error if you want the simulation to stop and display an error when the offset values are out of range.
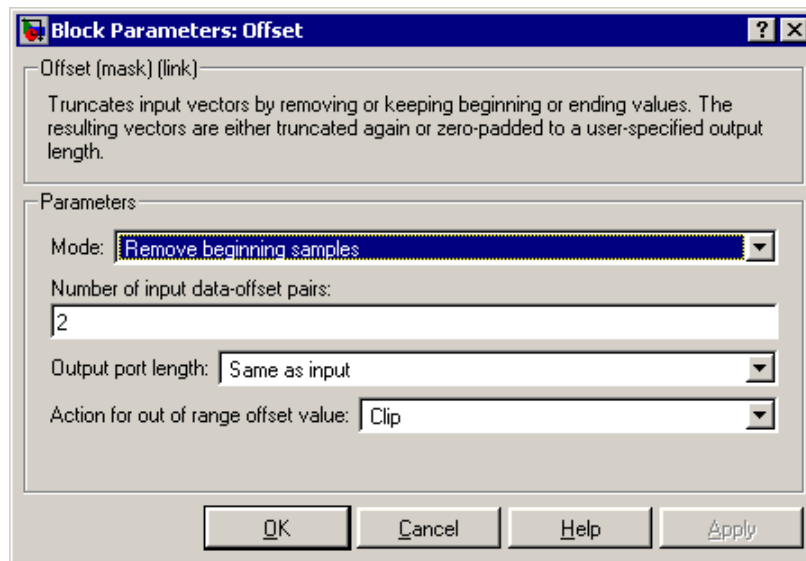
Dialog Box

Mode

Use this parameter to determine which values the block discards or retains from the input vector. Your choices are Remove beginning samples, Remove ending samples, Keep beginning samples, and Keep ending samples.

Number of input data-offset pairs

Specify the number of inputs to the block. The number of input ports is twice the scalar value you enter.

Output port length

Use this parameter to specify the length of the output vectors. If you select `Same as input`, the output vectors are the same length as the input vectors. If you select `User-defined`, you can enter the desired length of the output vectors.

Output length

Enter a scalar that represents the desired length of the output vectors. This parameter is visible if, for the **Output port length** parameter, you select `User-defined`.

Action for out of range offset value

Use this parameter to determine how the block behaves when an offset value is not in the range such that $0 \leq$ offset value $\leq N$, where N is the input vector length. When you want any offset values less than 0 to be set to 0 and any offset values greater than N to be set to N, select `Clip`. When you want to be warned when any offset values less than 0 are set to 0 and any offset values greater than N are set to N, select `Clip and warn`. When you want the simulation to stop and display an error when the offset values are out of range, select `Error`.

Offset

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
O	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

Purpose   Implement overlap-add method of frequency-domain filtering

Library   Filtering / Filter Implementations

dsparch4

Description

Overlap
Add

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.

The block accepts vector or matrix inputs, and treats each column of
the input as an individual channel. The block unbuffers the input data
into row vectors such that the length of the output vector is equal to the
number of channels in the input. The data output rate of the block is
M times faster than its data input rate, where *M* is the length of the
columns in the input (frame-size).

The block breaks the scalar input sequence u, of length nu, into length-L
nonoverlapping data sections,
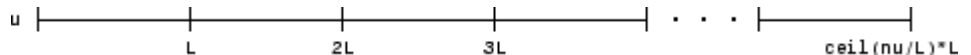
which it linearly convolves with the filter's FIR coefficients,

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_{n+1} z^{-n}$$

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 product, such as fir1. All filter states are
internally initialized to zero.

When either the filter coefficients or the inputs to the block are complex,
the **Output** parameter should be set to Complex. Otherwise, the default
Output setting, Real, instructs the block to take only the real part
of the solution.

The block's overlap-add operation is equivalent to

Overlap-Add FFT Filter

```
y = ifft(fft(u(i:i+L-1),nfft) .* fft(b,nfft))
```

where you specify nfft in the **FFT size** parameter as a power-of-two value greater (typically *much* greater) than n+1. Values for **FFT size** that are not powers of two are rounded upwards to the nearest power-of-two value to obtain nfft.

The block overlaps successive output sections by n points and sums them.
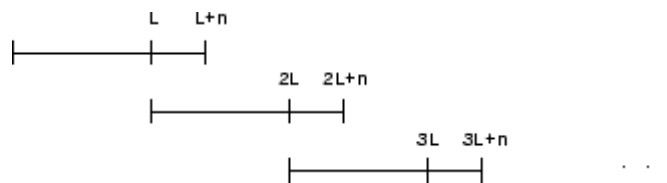
The first L samples of each summation are output in sequence. The block chooses the parameter L based on the filter order and the FFT size.

```
L = nfft - n
```

Latency

In *single-tasking* operation, the Overlap-Add FFT Filter block has a latency of nfft-n+1 samples. The first nfft-n+1 consecutive outputs from the block are zero; the first filtered input value appears at the output as sample nfft-n+2.

In *multitasking* operation, the Overlap-Add FFT Filter block has a latency of 2*(nfft-n+1) samples. The first 2*(nfft-n+1) consecutive outputs from the block are zero; the first filtered input value appears at the output as sample 2*(nfft-n)+3.

Note For more information on latency and the Simulink software tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the Simulink Coder documentation.
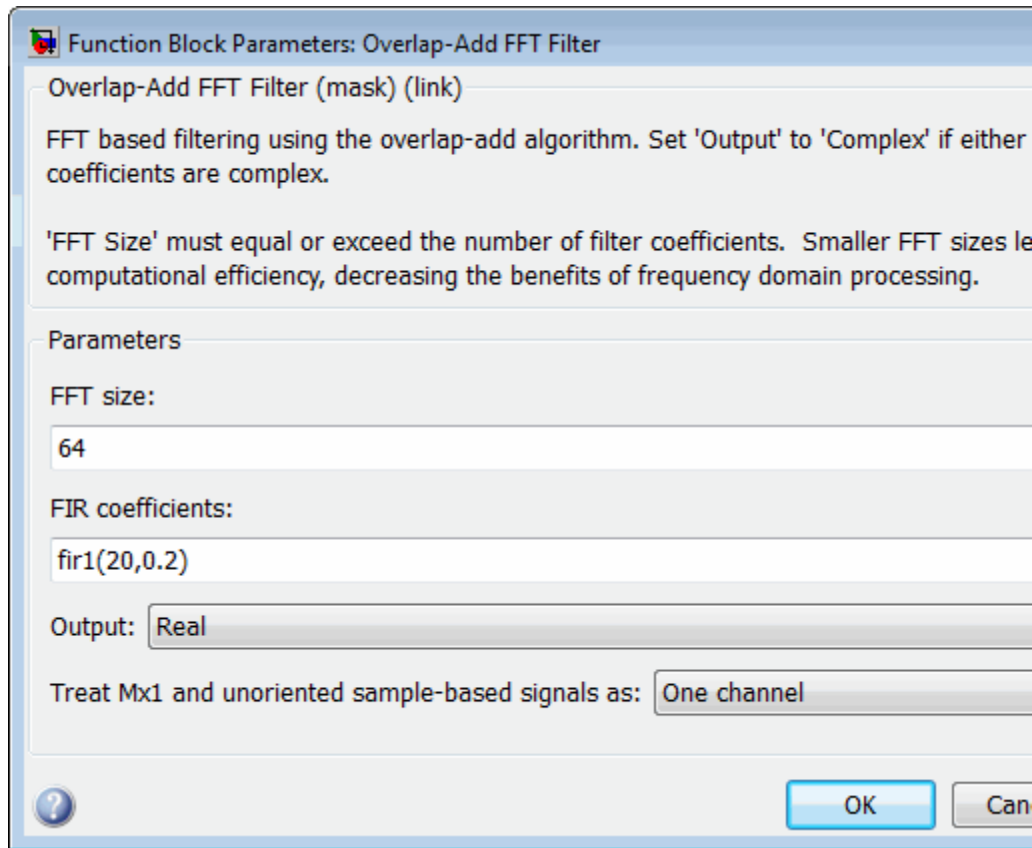
Dialog Box

FFT size

  The size of the FFT, which should be a power-of-two value greater than the length of the specified FIR filter.

FIR coefficients

  The filter numerator coefficients.

Overlap-Add FFT Filter

Output
> The complexity of the output; `Real` or `Complex`. When the input signal or the filter coefficients are complex, this should be set to `Complex`.

Treat Mx1 and unoriented sample-based signals as
> Specify how the block treats sample-based *M*-by-1 column vectors and unoriented sample-based vectors of length *M*. You can select one of the following options:
>
> - `One channel` — When you select this option, the block treats *M*-by-1 and unoriented sample-based inputs as a column vector (one channel).
>
> - `M channels (this choice will be removed   see release notes)` — When you select this option, the block treats *M*-by-1 and unoriented sample-based inputs as a 1-by-*M* row vector.

> **Note** This parameter will be removed in a future release. At that time, the block will always treat *M*-by-1 and unoriented vectors as a single channel.

References Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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

Supported Data Types
- Double-precision floating point
- Single-precision floating point

See Also Overlap-Save FFT Filter   DSP System Toolbox product

Purpose	Implement overlap-save method of frequency-domain filtering
Library	Filtering / Filter Implementations
	dsparch4

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.

The block accepts vector or matrix inputs, and treats each column of the input as an individual channel. The block unbuffers the input data into row vectors such that the length of the output vector is equal to the number of channels in the input. The data output rate of the block is *M* times faster than its data input rate, where *M* is the length of the columns in the input (frame-size).

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 product, such as fir1. All filter states are internally initialized to zero.

When either the filter coefficients or the inputs to the block are complex, the **Output** parameter should be set to Complex. Otherwise, the default **Output** setting, Real, instructs the block to take only the real part of the solution.

The 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))
```

Overlap-Save FFT Filter

where you specify nfft in the **FFT size** parameter as a power of two value greater (typically *much* greater) than n+1. Values for **FFT size** that are not powers of two are rounded upwards to the nearest power-of-two value to obtain nfft.

The first n points of the circular convolution are invalid and are discarded. The Overlap-Save FFT Filter block outputs the remaining nfft-n points, which are equivalent to the linear convolution.

Latency

In *single-tasking* operation, the Overlap-Save FFT Filter block has a latency of nfft-n+1 samples. The first nfft-n+1 consecutive outputs from the block are zero; the first filtered input value appears at the output as sample nfft-n+2.

In *multitasking* operation, the Overlap-Save FFT Filter block has a latency of 2*(nfft-n+1) samples. The first 2*(nfft-n+1) consecutive outputs from the block are zero; the first filtered input value appears at the output as sample 2*(nfft-n)+3.

Note For more information on latency and the Simulink environment tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the Simulink Coder documentation.

Dialog Box

FFT size
    The size of the FFT, which should be a power of two value greater than the length of the specified FIR filter.

FIR coefficients
    The filter numerator coefficients.

Output

The complexity of the output; `Real` or `Complex`. When the input signal or the filter coefficients are complex, this should be set to `Complex`.

Treat Mx1 and unoriented sample-based signals as

Specify how the block treats sample-based M-by-1 column vectors and unoriented sample-based vectors of length M. You can select one of the following options:

- `One channel` — When you select this option, the block treats M-by-1 and unoriented sample-based inputs as a column vector (one channel).

- `M channels (this choice will be removed — see release notes)` — When you select this option, the block treats M-by-1 and unoriented sample-based inputs as a 1-by-M row vector.

Note This parameter will be removed in a future release. At that time, the block will always treat M-by-1 and unoriented vectors as a single channel.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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

Supported Data Types

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

See Also

Overlap-Add FFT Filter	DSP System Toolbox

Purpose   Overwrite submatrix or subdiagonal of input

Library   • Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

• Signal Management / Indexing

dspindex
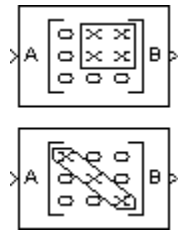
Description   The Overwrite Values block overwrites a contiguous submatrix or subdiagonal of an input matrix. You can provide the overwriting values by typing them in a block parameter, or through an additional input port, which is useful for providing overwriting values that change at each time step.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \Rightarrow \quad \Rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \end{bmatrix}$$
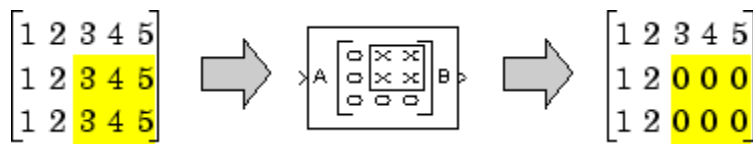
The block accepts scalars, vectors and matrices. The output always has the same size as the original input signal, not necessarily the same size as the signal containing the overwriting values. The input(s) and output of this block must have the same data type.

Specifying the Overwriting Values

The **Source of overwriting value(s)** parameter determines how you must provide the overwriting values, and has the following settings.

• Specify via dialog — You must provide the overwriting value(s) in the **Overwrite with** parameter. The block uses the same overwriting values to overwrite the specified portion of the input at each time step. To learn how to specify valid overwriting values, see "Valid Overwriting Values" on page 2-1160.

• Second input port — You must provide overwriting values through a second block input port, V. Use this setting to provide different

2-1159
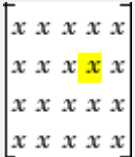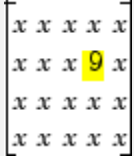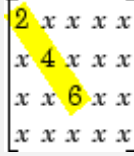
Overwrite Values

overwriting values at each time step. The output inherits its size and rate from the input signal, *not* the overwriting values.

The rate at which you provide the overwriting values through input port V must match the rate at which the block receives each input matrix at input port A. In other words, the input signals must have the same Simulink sample time.
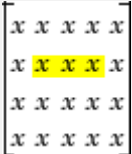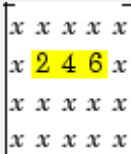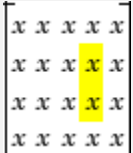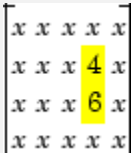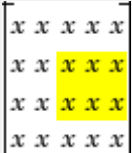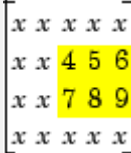
Valid Overwriting Values

The overwriting values can be a single constant, vector, or matrix, depending on the portion of the input you are overwriting, regardless of whether you provide the overwriting values through an input port or by providing them in the **Overwrite with** parameter.

Valid Overwriting Values

Portion of Input to Overwrite	Valid Overwriting Values	Example
A single element in the input $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any constant value, v	$v = 9$ $\begin{bmatrix} x & x & x & x & x \\ x & x & x & 9 & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$
A length-k portion of the diagonal $\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}$	Any length-k column or row vector, v	$k = 3 \quad v = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$ $\begin{bmatrix} 2 & x & x & x \\ x & 4 & x & x \\ x & x & 6 & x & x \\ x & x & x & x \end{bmatrix}$

Valid Overwriting Values (Continued)

Portion of Input to Overwrite	Valid Overwriting Values	Example
A length-k portion of a row $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}$	Any length-k row vector, v	$k = 3 \qquad v = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & 2 & 4 & 6 & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$
A length-k portion of a column $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any length-k column vector, v	$k = 2 \qquad v = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & x & x & 4 & x \\ x & x & x & 6 & x \\ x & x & x & x & x \end{bmatrix}$
An m-by-n submatrix $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any m-by-n matrix, v	$m = 2$ $n = 3$ $\qquad v = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & x & 4 & 5 & 6 \\ x & x & 7 & 8 & 9 \\ x & x & x & x & x \end{bmatrix}$

This block supports Simulink virtual buses.

Overwrite Values

Block Parameters: Overwrite Values

Overwrite Values (mask) (link)

Overwrites a selected portion of the input matrix--either a submatrix, full diagonal, or a portion of the diagonal.
Specify overwriting values as follows:
   --Matrix with the same dimensions as the submatrix
   --Vector with the same length as the portion of the diagonal
   --Scalar constant with which to replace each element in the submatrix or diagonal portion.

Treats unoriented (1-D) input vectors as column vectors.

Parameters

Overwrite: Submatrix

Source of overwriting value(s): Specify via dialog

Overwrite with:
0

Row span: Range of rows

Starting row: First

Starting row index:
1

Ending row: Last

Ending row index:
1

Column span: Range of columns

Starting column: First

Starting column index:
1

Ending column: Last

Ending column index:
1

Dialog Box

Note Only some of the following parameters are visible in the dialog box at any one time.

Overwrite

Determines whether to overwrite a specified submatrix or a specified portion of the diagonal.

Source of overwriting value(s)

Determines where you must provide the overwriting values: either through an input port, or by providing them in the **Overwrite with** parameter. For more information, see "Specifying the Overwriting Values" on page 2-1159.

Overwrite with

The value(s) with which to overwrite the specified portion of the input matrix. Enabled only when **Source of overwriting value(s)** is set to Specify via dialog. To learn how to specify valid overwriting values, see "Valid Overwriting Values" on page 2-1160.

Row span

The range of input rows to be overwritten. Options are All rows, One row, or Range of rows. For descriptions of these options, see "Dialog Box" on page 2-1162.

Row/Starting row

The input row that is the first row of the submatrix that the block overwrites. For a description of the options for the **Row** and **Starting row** parameters, see Settings for Row, Column, Starting Row, and Starting Column Parameters on page 2-1168. **Row** is enabled when **Row span** is set to One row, and **Starting row** when **Row span** is set to Range of rows.

Row index/Starting row index

Index of the input row that is the first row of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters on page 2-1168. **Row index** is enabled when **Row** is set to Index, and **Starting row index** when **Starting row** is set to Index.

Row offset/Starting row offset

The offset of the input row that is the first row of the submatrix that the block overwrites. See how to use these parameters in

Overwrite Values

Settings for Row, Column, Starting Row, and Starting Column Parameters on page 2-1168. **Row offset** is enabled when **Row** is set to `Offset from middle` or `Offset from last`, and **Starting row offset** is enabled when **Starting row** is set to `Offset from middle` or `Offset from last`.

Ending row

The input row that is the last row of the submatrix that the block overwrites. For a description of this parameter's options, see Settings for Ending Row and Ending Column Parameters on page 2-1169. This parameter is enabled when **Row span** is set to `Range of rows`, and **Starting row** is set to any option but `Last`.

Ending row index

Index of the input row that is the last row of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters on page 2-1169. Enabled when **Ending row** is set to `Index`.

Ending row offset

The offset of the input row that is the last row of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters on page 2-1169. Enabled when **Ending row** is set to `Offset from middle` or `Offset from last`.

Column span

The range of input columns to be overwritten. Options are `All columns`, `One column`, or `Range of columns`. For descriptions of the analogous row options, see "Dialog Box" on page 2-1162.

Column/Starting column

The input column that is the first column of the submatrix that the block overwrites. For a description of the options for the **Column** and **Starting column** parameters, see Settings for Row, Column, Starting Row, and Starting Column Parameters on page 2-1168. **Column** is enabled when **Column span** is set to `One column`, and **Starting column** when **Column span** is set to `Range of columns`.

Column index/Starting column index

Index of the input column that is the first column of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters on page 2-1168. **Column index** is enabled when **Column** is set to `Index`, and **Starting column index** when **Starting column** is set to `Index`.

Column offset/Starting column offset

The offset of the input column that is the first column of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters on page 2-1168. **Column offset** is enabled when **Column** is set to `Offset from middle` or `Offset from last`, and **Starting column offset** is enabled when **Starting column** is set to `Offset from middle` or `Offset from last`.

Ending column

The input column that is the last column of the submatrix that the block overwrites. For a description of this parameter's options, see Settings for Ending Row and Ending Column Parameters on page 2-1169. This parameter is enabled when **Column span** is set to `Range of columns`, and **Starting column** is set to any option but `Last`.

Ending column index

Index of the input column that is the last column of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters on page 2-1169. This parameter is enabled when **Ending column** is set to `Index`.

Ending column offset

The offset of the input column that is the last column of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters on page 2-1169. This parameter is enabled when

Ending column is set to `Offset from middle` or `Offset from last`.

Diagonal span

The range of diagonal elements to be overwritten. Options are `All elements`, `One element`, or `Range of elements`. For descriptions of these options, see "Overwriting a Subdiagonal" on page 2-1172.

Element/Starting element

The input diagonal element that is the first element in the subdiagonal that the block overwrites. For a description of the options for the **Element** and **Starting element** parameters, see Element and Starting Element Parameters on page 2-1172. **Element** is enabled when **Element span** is set to `One element`, and **Starting element** when **Element span** is set to `Range of elements`.

Element index/Starting element index

Index of the input diagonal element that is the first element of the subdiagonal that the block overwrites. See how to use these parameters in Element and Starting Element Parameters on page 2-1172. **Element index** is enabled when **Element** is set to `Index`, and **Starting element index** when **Starting element** is set to `Index`.

Element offset/Starting element offset

The offset of the input diagonal element that is the first element of the subdiagonal that the block overwrites. See how to use these parameters in Element and Starting Element Parameters on page 2-1172. **Element offset** is enabled when **Element** is set to `Offset from middle` or `Offset from last`, and **Starting element offset** is enabled when **Starting element** is set to `Offset from middle` or `Offset from last`.

Ending element

The input diagonal element that is the last element of the subdiagonal that the block overwrites. For a description of this parameter's options, see Ending Element Parameters on page 2-1173. This parameter is enabled when **Element span** is set to

Range of elements, and **Starting element** is set to any option but Last.

Ending element index
Index of the input diagonal element that is the last element of the subdiagonal that the block overwrites. See how to use this parameter in Ending Element Parameters on page 2-1173. This parameter is enabled when **Ending element** is set to Index.

Ending element offset
The offset of the input diagonal element that is the last element of the subdiagonal that the block overwrites. See how to use this parameter in Ending Element Parameters on page 2-1173. This parameter is enabled when **Ending element** is set to Offset from middle or Offset from last.

Examples

Overwriting a Submatrix

To overwrite a submatrix, follow these steps:

1 Set the **Overwrite** parameter to Submatrix.

2 Specify the overwriting values as described in "Specifying the Overwriting Values" on page 2-1159.

3 Specify which rows and columns of the input matrix are contained in the submatrix that you want to overwrite by setting the **Row span** parameter to one of the following options and the **Column span** to the analogous column-related options:

- All rows — The submatrix contains all rows of the input matrix.

- One row — The submatrix contains only one row of the input matrix, which you must specify in the **Row** parameter, as described in the following table.

- Range of rows — The submatrix contains one or more rows of the input, which you must specify in the **Starting Row** and **Ending row** parameters, as described in the following tables.

Overwrite Values

4 When you set **Row span** to `One row` or `Range of rows`, you need to further specify the row(s) contained in the submatrix by setting the **Row** or **Starting row** and **Ending row** parameters. Likewise, when you set **Column span** to `One column` or `Range of columns`, you must further specify the column(s) contained in the submatrix by setting the **Column** or **Starting column** and **Ending column** parameters. For descriptions of the settings for these parameters, see the following tables.

Settings for Row, Column, Starting Row, and Starting Column Parameters

Settings for Specifying the Submatrix's First Row or Column	First Row of Submatrix (Only row for Row span = One row)	First Column of Submatrix (Only row for Row span = One row)
`First`	First row of the input	First column of the input
`Index`	Input row specified in the **Row index** parameter	Input column specified in the **Column index** parameter
`Offset from last`	Input row with the index $M - \texttt{rowOffset}$ where M is the number of input rows, and `rowOffset` is the value of the **Row offset** or **Starting row offset** parameter	Input column with the index $N - \texttt{colOffset}$ where N is the number of input columns, and `colOffset` is the value of the **Column offset** or **Starting column offset** parameter
`Last`	Last row of the input	Last column of the input

Settings for Row, Column, Starting Row, and Starting Column Parameters (Continued)

Settings for Specifying the Submatrix's First Row or Column	First Row of Submatrix (Only row for Row span = One row)	First Column of Submatrix (Only row for Row span = One row)
Offset from middle	Input row with the index $\text{floor}(M/2 + 1 - \text{rowOffset})$ where M is the number of input rows, and rowOffset is the value of the **Row offset** or **Starting row offset** parameter	Input column with the index $\text{floor}(N/2 + 1 - \text{rowOffset})$ where N is the number of input columns, and colOffset is the value of the or **Column offset** or **Starting column offset** parameter
Middle	Input row with the index $\text{floor}(M/2 + 1)$ where M is the number of input rows	Input columns with the index $\text{floor}(N/2 + 1)$ where N is the number of input columns

Settings for Ending Row and Ending Column Parameters

Settings for Specifying the Submatrix's Last Row or Column	Last Row of Submatrix	Last Column of Submatrix
Index	Input row specified in the **Ending row index** parameter	Input column specified in the **Ending column index** parameter
Offset from last	Input row with the index $M - \text{rowOffset}$ where M is the number of input	Input column with the index $N - \text{colOffset}$ where N is the number of input

Overwrite Values

Settings for Ending Row and Ending Column Parameters (Continued)

Settings for Specifying the Submatrix's Last Row or Column	Last Row of Submatrix	Last Column of Submatrix
	rows, and rowOffset is the value of the **Ending row offset** parameter	columns, and colOffset is the value of the **Ending column offset** parameter
Last	Last row of the input	Last column of the input
Offset from middle	Input row with the index floor(M/2 + 1 - rowOffset) where M is the number of input rows, and rowOffset is the value of the **Ending row offset** parameter	Input column with the index floor(N/2 + 1 - rowOffset) where N is the number of input columns, and colOffset is the value of the **Ending column offset** parameter
Middle	Input row with the index floor(M/2 + 1) where M is the number of input rows	Input columns with the index floor(N/2 + 1) where N is the number of input columns

For example, to overwrite the lower-right 2-by-3 submatrix of a 3-by-5 input matrix with all zeros, enter the following set of parameters:

- **Overwrite** = Submatrix

- **Source of overwriting value(s)** = Specify via dialog

- **Overwrite with** = 0

- **Row span** = Range of rows

- **Starting row** = Index

- **Starting row index** = 2

- **Ending row** = Last

- **Column span** = `Range of columns`
- **Starting column** = `Offset from last`
- **Starting column offset** = `2`
- **Ending column** = `Last`

The following figure shows the block with the above settings overwriting a portion of a 3-by-5 input matrix.
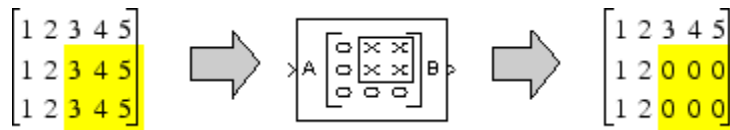
There are often several possible parameter combinations that select the *same* submatrix from the input. For example, instead of specifying `Last` for **Ending column**, you could select the same submatrix by specifying

- **Ending column** = `Index`
- **Ending column index** = `5`

Overwrite Values

Overwriting a Subdiagonal

To overwrite a subdiagonal, follow these steps:

1 Set the **Overwrite** parameter to `Diagonal`.

2 Specify the overwriting values as described in "Specifying the Overwriting Values" on page 2-1159.

3 Specify the subdiagonal that you want to overwrite by setting the **Diagonal span** parameter to one of the following options:

- `All elements` — Overwrite the entire input diagonal.

- `One element` — Overwrite one element in the diagonal, which you must specify in the **Element** parameter (described below).

- `Range of elements` — Overwrite a portion of the input diagonal, which you must specify in the **Starting element** and **Ending element** parameters, as described in the following table.

4 When you set **Diagonal span** to `One element` or `Range of elements`, you need to further specify which diagonal element(s) to overwrite by setting the **Element** or **Starting element** and **Ending element** parameters. See the following tables.

Element and Starting Element Parameters

Settings for Element and Starting Element Parameters	First Element in Subdiagonal (Only element when Diagonal span = One element)
First	Diagonal element in first row of the input
Index	kth diagonal element, where k is the value of the **Element index** or **Starting element index** parameter

Element and Starting Element Parameters (Continued)

Settings for Element and Starting Element Parameters	First Element in Subdiagonal (Only element when Diagonal span = One element)
Offset from last	Diagonal element in the row with the index `M - offset` where M is the number of input rows, and offset is the value of the **Element offset** or **Starting element offset** parameter
Last	Diagonal element in the last row of the input
Offset from middle	Diagonal element in the input row with the index `floor(M/2 + 1 - offset)` where M is the number of input rows, and offset is the value of the **Element offset** or **Starting element offset** parameter
Middle	Diagonal element in the input row with the index `floor(M/2 + 1)` where M is the number of input rows

Ending Element Parameters

Settings for Ending Element Parameter	Last Element in Subdiagonal
Index	kth diagonal element, where k is the value of the **Ending element index** parameter
Offset from last	Diagonal element in the row with the index `M - offset` where M is the number of input rows, and offset is the value of the **Ending element offset** parameter
Last	Diagonal element in the last row of the input

Overwrite Values

Ending Element Parameters (Continued)

Settings for Ending Element Parameter	Last Element in Subdiagonal
Offset from middle	Diagonal element in the input row with the index $floor(M/2 + 1 - offset)$ where M is the number of input rows, and offset is the value of the **Ending element offset** parameter
Middle	Diagonal element in the input row with the index $floor(M/2 + 1)$ where M is the number of input rows

Supported Data Types

The input(s) and output of this block must have the same data type.

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
V	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers
B	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Reshape	Simulink
Selector	Simulink
Submatrix	DSP System Toolbox
Variable Selector	DSP System Toolbox
reshape	MATLAB

Pad

Purpose   Pad or truncate specified dimension(s)

Library   Signal Operations

dspsigops

Description   The Pad block extends or crops the dimensions of the input by padding or truncating along its columns, rows, columns and rows, or any dimension(s) you specify. Truncation occurs when you specify output dimensions that are shorter than the corresponding input dimensions. If the input and output lengths are the same, the block is a pass-through.

You can enter the pad value in the block mask or via an input port. You can enter output sizes in the block mask, or have the block pad the specified dimensions until their length is the next highest power of two. The **Pad signal at** parameter controls whether the specified input dimensions are padded or truncated at their beginning, end, or both. For odd pad or truncation lengths, the extra pad value or truncation is applied to the end of the signal. When the block is in Specified dimensions mode, you can specify either the output size or the pad size.

You can have the block warn or error when an input signal is truncated using the **Action when truncation occurs** parameter.

**Dialog
Box**
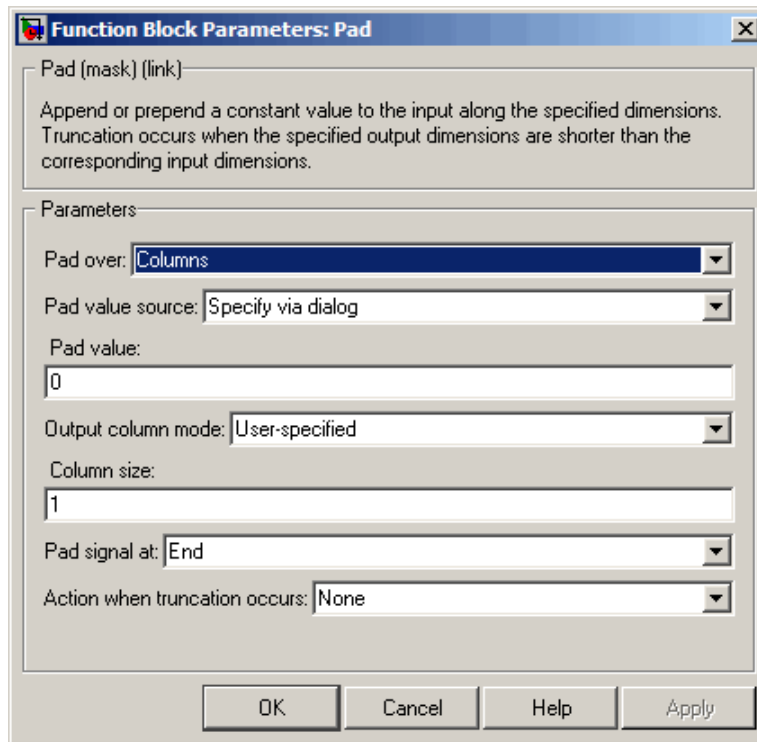
Pad over

Specify the dimensions over which to pad or truncate: Columns,
Rows, Columns and rows, None, or Specified dimensions.

Dimensions to pad

Specify the one-based dimension(s) over which to pad or truncate.
The value for this parameter can be a scalar or a vector. For
example, specify 1 to pad columns. Specify [1 2] to pad columns
and rows. Specify [1 3 5] to pad the first, third, and fifth
dimensions.

This parameter is only visible when Specified dimensions is
selected for the **Pad over** parameter.

Pad

Pad value source

Choose how you specify the pad value. The pad value can come from an input port or from the dialog:

- If you select `Input port`, the `PVal` port appears on the block icon.

- If you select `Specify via dialog`, the **Pad value** parameter appears.

Pad value

Specify the constant scalar value with which to pad the input. Tunable.

This parameter is only visible when `Specify via dialog` is selected for the **Pad value source** parameter.

Output column mode

Choose how you specify the column length of the output:

- If you select `User-specified`, the **Column size** parameter appears.

- If you select `Next power of two`, the block pads the output columns until their length is the next highest power of two. If the column length is already a power of two, the columns are not padded.

This parameter is only visible when `Columns` or `Columns and rows` is selected for the **Pad over** parameter.

Column size

Specify the column length of the output. If the specified column length is longer than the input column length, the columns are padded. If the specified column length is shorter than the input column length, the columns are truncated. This parameter is only visible when `User-specified` is selected for the **Output column mode** parameter.

Output row mode

Choose how you specify the output row length of the output:

- If you select `User-specified`, the **Row size** parameter appears.

- If you select `Next power of two`, the block pads the output rows until their length is the next highest power of two. If the row length is already a power of two, the rows are not padded.

This parameter is only visible when `Rows` or `Columns and rows` is selected for the **Pad over** parameter.

Row size
Specify the row length of the output. If the specified row length is longer than the input row length, the rows are padded. If the specified row length is shorter than the input row length, the rows are truncated. This parameter is only visible when `User-specified` is selected for the **Output row mode** parameter.

Specify
Choose whether you want to control the output length of the specified dimensions by specifying the pad size or the output size.

This parameter is only visible when `Specified dimensions` is selected for the **Pad over** parameter.

Pad size at beginning
Specify how many values to add to the beginning of the input signal along the specified dimension(s). This parameter must be a scalar or a vector with the same number of elements as the **Dimensions to pad** parameter. Each element in the **Pad size at beginning** parameter gives the pad length for the beginning of the corresponding dimension in the **Dimensions to pad** parameter. Values of this parameter must be zero or a positive integer.

This parameter is only visible if `Pad size` is selected for the **Specify** parameter.

Pad

Pad size at end

Specify how many values to add to the end of the input signal along the specified dimension(s). This parameter must be a scalar or a vector with the same number of elements as the **Dimensions to pad** parameter. Each element in the **Pad size at end** parameter gives the pad length for the end of the corresponding dimension in the **Dimensions to pad** parameter. Values of this parameter must be zero or a positive integer.

This parameter is only visible if `Pad size` is selected for the **Specify** parameter.

Output size mode

Choose how you specify the output length of the specified dimensions:

- If you select `User-specified`, the **Output size** parameter appears.

- If you select `Next power of two`, the block pads the specified dimensions until their length is the next highest power of two. If the dimension length is already a power of two, no padding occurs in that dimension.

This parameter is only visible if `Output size` is selected for the **Specify** parameter.

Output size

Specify the output length of the specified dimension(s). This parameter must be a scalar or a vector with the same number of elements as the **Dimensions to pad** parameter. Each element in the **Output size** vector gives the output length for the corresponding dimension in the **Dimensions to pad** vector. If the specified length is longer than the input length for a given dimensions, that dimension is padded. If the specified length is shorter than the input length for a given dimension, that dimension is truncated.

This parameter is only visible if `Output size` is selected for the **Specify** parameter.

Pad signal at

Specify whether to pad or truncate the signal at the `Beginning`, `End`, or `Beginning and end` of the specified dimension(s). When you select `Beginning and end`, half the pad length is added to the beginning of the signal, and half is added to the end of the signal. For an odd pad length, the extra value is added to the end of the signal. This also applies to truncation. In this mode, an equal number of values are truncated from the beginning and the end of the signal. In the case of an odd truncation length, the extra value is removed from the end of the signal.

Action when truncation occurs

Choose `None` when you do not want to be notified that the input is truncated. Select `Warning` to display a warning when the input is truncated. Choose `Error` when to display an error and terminate the simulation when the input is truncated.

Supported Data Types	**Port**	**Supported Data Types**
	Input	• Double-precision floating-point
		• Single-precision floating-point
		• Fixed point (signed and unsigned)
		• Boolean
		• 8-, 16-, and 32-bit signed integers
		• 8-, 16-, and 32-bit unsigned integers
	Output	• Double-precision floating-point
		• Single-precision floating-point
		• Fixed point (signed and unsigned)
		• Boolean

Pad

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Concatenate	Simulink
Repeat	DSP System Toolbox
Submatrix	DSP System Toolbox
Upsample	DSP System Toolbox
Variable Selector	DSP System Toolbox

Purpose   Design parametric equalizer

Library   Filtering / Filter Designs

                     dspfdesign

Description   This block brings the filter design capabilities of the `filterbuilder` function to the Simulink environment.
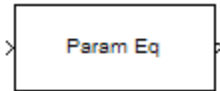
> Param Eq >

Dialog Box   See "Parametric Equalizer Filter Design Dialog Box — Main Pane" on page 7-761 for more information about the parameters of this block. The **Data Types** and **Code Generation** panes are not available for blocks in the DSP System Toolbox Filter Designs library.

                     Parameters of this block that do not change filter order or structure are tunable.

Parametric Equalizer

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

Order mode

Select `Minimum` to design a minimum order filter that meets the design specifications, or `Specify` to enter a specific filter order. The order mode also affects the possible frequency constraints, which in turn limit the gain specifications. For example, if you specify a `Minimum` order filter, the available frequency constraints are:

- `Center frequency, bandwidth, passband width`

- `Center frequency, bandwidth, stopband width`

If you select `Specify`, the available frequency constraints are:

- `Center frequency, bandwidth`

- `Center frequency, quality factor`

- `Shelf type, cutoff frequency, quality factor`

- `Shelf type, cutoff frequency, shelf slope parameter`

- `Low frequency, high frequency`

Parametric Equalizer

Order

> Specify the filter order. This parameter is enabled only if the **Order mode** is set to `Specify`.

Frequency specifications

Depending on the filter order, the possible frequency constraints change. Once you choose the frequency constraints, the input boxes in this area change to reflect the selection.

Frequency constraints

> Select the specification to represent the frequency constraints. The following options are available:
>
> - `Center frequency, bandwidth, passband width` (available for minimum order only)
>
> - `Center frequency, bandwidth, stopband width` (available for minimum order only)
>
> - `Center frequency, bandwidth` (available for a specified order only)
>
> - `Center frequency, quality factor` (available for a specified order only)
>
> - `Shelf type, cutoff frequency, quality factor` (available for a specified order only)
>
> - `Shelf type, cutoff frequency, shelf slope parameter` (available for a specified order only)
>
> - `Low frequency, high frequency` (available for a specified order only)

Frequency units

> Select the frequency units from the available drop down list (`Normalized, Hz, kHz, MHz, GHz`). If `Normalized` is selected, then the **Input Fs** box is disabled for input.

Input Fs

Enter the input sampling frequency. This input box is disabled for input if `Normalized` is selected in the **Frequency units** input box.

Center frequency

Enter the center frequency in the units specified by the value in **Frequency units**.

Bandwidth

The bandwidth determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Bandwidth gain** in the **Gain specifications** section. By default, the **Bandwidth gain** defaults to `db(sqrt(.5))`, or –3 dB relative to the center frequency. The **Bandwidth** property only applies when the **Frequency constraints** are: `Center frequency, bandwidth, passband width`, `Center frequency, bandwidth, stopband width`, or `Center frequency, bandwidth`.

Passband width

The passband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Passband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, passband width`.

Stopband width

The stopband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Stopband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, stopband width`.

Low frequency

Enter the low frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected

is `Low frequency, high frequency`. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

High frequency

Enter the high frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

Gain Specifications

Depending on the filter order and frequency constraints, the possible gain constraints change. Also, once you choose the gain constraints the input boxes in this area change to reflect the selection.

Gain constraints

Select the specification array to represent gain constraints, and remember that not all of these options are available for all configurations. The following is a list of all available options:

- `Reference, center frequency, bandwidth, passband`

- `Reference, center frequency, bandwidth, stopband`

- `Reference, center frequency, bandwidth, passband, stopband`

- `Reference, center frequency, bandwidth`

Gain units

Specify the gain units either `dB` or `squared`. These units are used for all gain specifications in the dialog box.

Reference gain

The reference gain determines the level to which the filter magnitude attenuates in **Gain units**. The reference gain is a *floor* gain for the filter magnitude response. For example, you may use the reference gain together with the **Center frequency gain** to leave certain frequencies unattenuated (reference gain of 0 dB) while boosting other frequencies.

Bandwidth gain

>Specifies the gain in **Gain units** at which the bandwidth is defined. This property applies only when the **Frequency constraints** specification contains a `bandwidth` parameter, or is `Low frequency, high frequency`.

Center frequency gain

>Specify the center frequency in **Gain units**

Passband gain

>The passband gain determines the level in **Gain units** at which the passband is defined. The passband is determined either by the **Passband width** value, or the **Low frequency** and **High frequency** values in the **Frequency specifications** section.

Stopband gain

>The stopband gain is the level in **Gain units** at which the stopband is defined. This property applies only when the **Order mode** is minimum and the **Frequency constaints** are `Center frequency, bandwidth, stopband width`.

Boost/cut gain

>The boost/cut gain applies only when the designing a shelving filter. Shelving filters include the `Shelf type` parameter in the **Frequency constraints** specification. The gain in the passband of the shelving filter is increased by **Boost/cut gain** dB from a *floor* gain of 0 dB.

Algorithm

Design method

>Select the design method from the drop-down list. Different methods are available depending on the chosen filter constraints.

Scale SOS filter coefficients to reduce chance of overflow

>Select the check box to scale the filter coefficients.

Parametric Equalizer

Filter Implementation

Structure

Specify filter structure. Choose from:

- Direct-form I SOS
- Direct-form II SOS
- Direct-form I transposed SOS
- Direct-form II transposed SOS

Use basic elements to enable filter customization

Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.
- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.
- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.
- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Note The `Inherited (this choice will be removed see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the sample rate of the input.

- `Allow multirate processing` — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to `Elements as channels (sample based)`.

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows

tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

Purpose
Determine whether each value of input signal is local minimum or maximum

Library
Signal Operations

dspsigops

Description
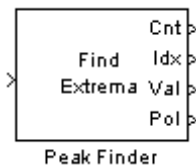
Peak Finder

The Peak Finder block counts the number of local extrema in each column of the real-valued input signal. The block outputs the number of local extrema at the Cnt port. You can also configure the block to output the extrema indices, the extrema values, and a binary indicator of whether or not the extrema are maxima or minima.

To qualify as an extrema, a point has to be larger (or smaller) than both of its neighboring points. Thus, end points are never considered extrema.

If you select the **Output peak indices** check box, the Idx port appears on the block. The block outputs the zero-based extrema indices at the Idx port. If you select the **Output peak values** check box, the Val port appears on the block. The block outputs the extrema values at the Val port. If you select either of these check boxes and set the **Peak type(s)** to Maxima and Minima, the Pol port also appears on the block. If the signal value is a maximum, the block outputs a 1 at the Pol ("Polarity") port. If the signal value is a minimum, the block outputs a 0 at the Pol port.

Use the **Maximum number of peaks to find** parameter to specify how many extrema to look for in each input signal. The block stops searching the input signal once this maximum number of extrema has been found.

If you select the **Ignore peaks within threshold of neighboring values** check box, the block no longer detects low-amplitude peaks. This feature allows the block to ignore noise within a threshold value that you define. Enter a threshold value for the **Threshold** parameter. Now, the current value is a maximum if (current – previous) > threshold and (current – next) > threshold. The current value is a minimum if (current – previous) < –threshold and (current – next) < –threshold.

Peak Finder

Examples   **Example 1**

Consider the input vector

`[9 6 10 3 4 5 0 12]`

The table below shows the analysis made by the Peak Finder block. Note that the first and last input signal values are not considered:

Previous, current, and next values	9 6 10	6 10 3	10 3 4	3 4 5	4 5 0	5 0 12
Current value if it is an extremum	6	10	3	—	5	0
Index of current value if it is an extremum	1	2	3	—	5	6
Polarity of current value if it is an extremum	0	1	0	—	1	0

Therefore, for this example the outputs at the block ports are

Cnt: `5`

Idx: `[1 2 3 5 6]`

Val: `[6 10 3 5 0]`

Pol: `[0 1 0 1 0]`

Example 2

Note that the **Overflow mode** parameter can affect the output of the block when the input is fixed point. Consider the following model:
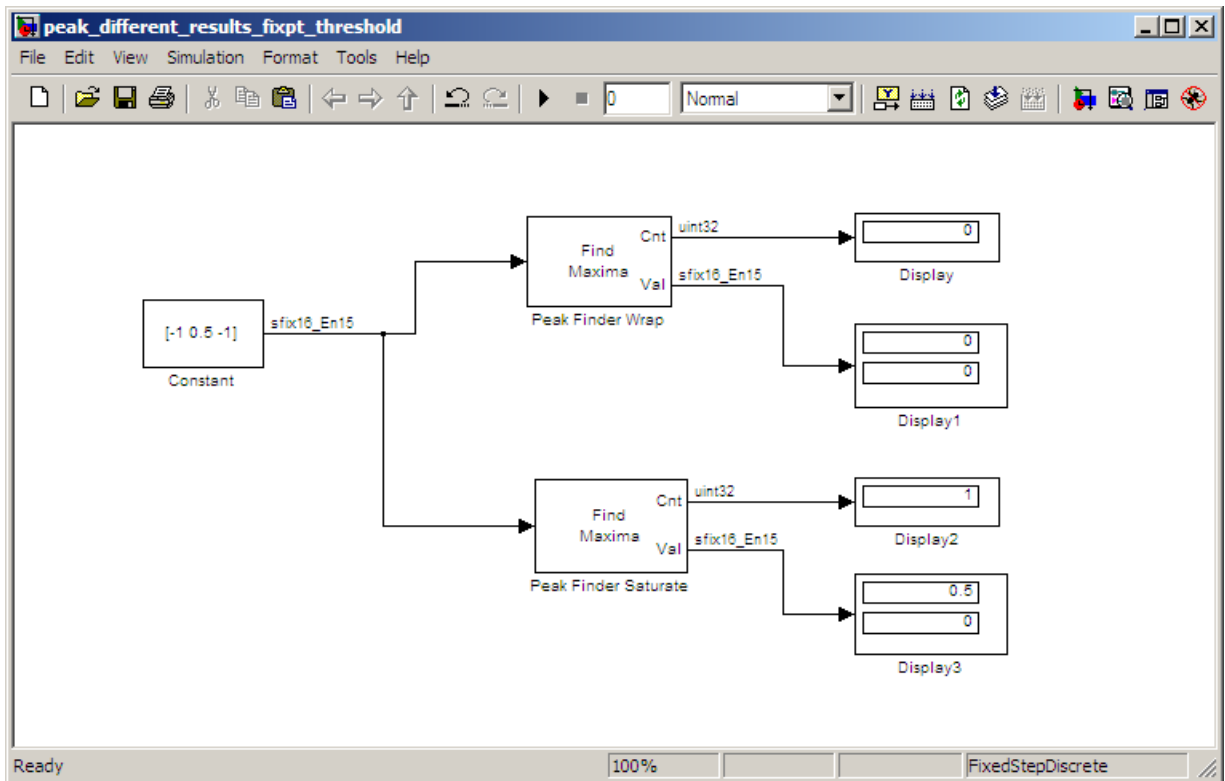
In this model, the settings in the Constant block are

- **Constant value** — [-1 0.5 -1]
- **Interpret vector parameters as 1–D** — not selected
- **Sampling mode** — Sample based
- **Sample time** — 1
- **Output data type** — <data type expression>
- **Mode** — Fixed point
- **Sign** — Signed

- **Scaling** — `Binary point`
- **Word length** — `16`
- **Fraction length** — `15`

The settings in the Peak Finder blocks are

- **Peak type(s)** — `Maxima`
- **Output peak indices** — not selected
- **Output peak values** — selected
- **Maximum number of peaks to find** — `2`
- **Ignore peaks within threshold of neighboring values** — selected
- **Threshold** — `0.25`
- **Overflow mode** — `Wrap` for Peak Finder Wrap, `Saturate` for Peak Finder Saturate

Setting the **Overflow mode** parameter of the Peak Finder Wrap block to `Wrap` causes the calculations (current – previous) > threshold and (current – next) > threshold to wrap on overflow, thereby causing the maximum to be missed.

Dialog Box

The **Main** pane of the Peak Finder block dialog appears as follows.
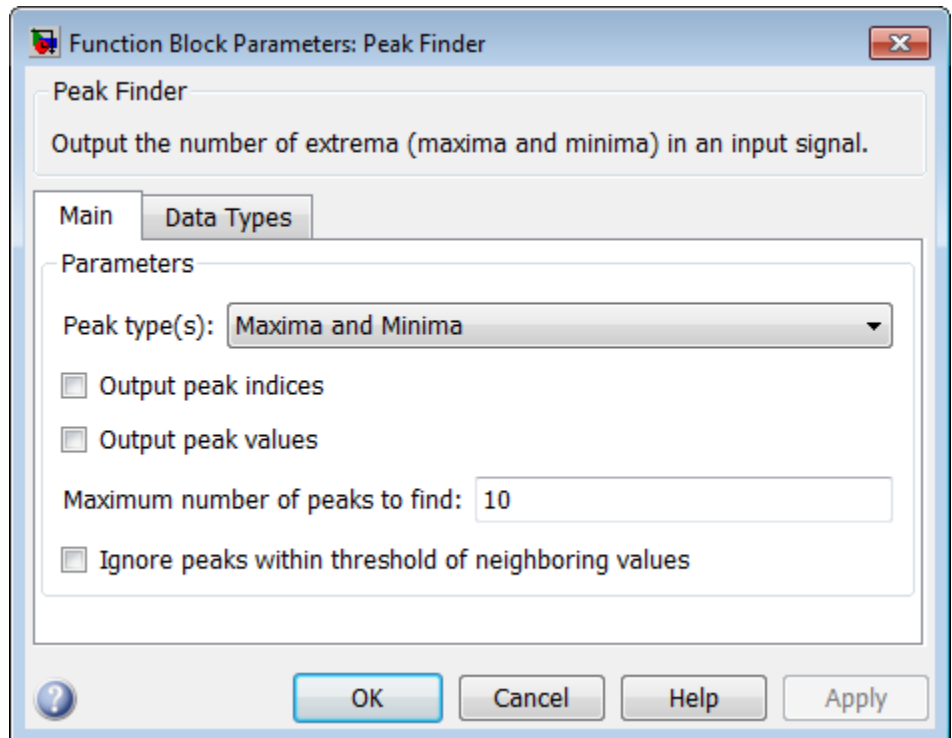
Peak type(s)

Specify whether you are looking for maxima, minima, or both.

Output peak indices

Select this check box if you want the block to output the extrema indices at the Idx port.

Output peak values

Select this check box if you want the block to output the extrema values at the Val port.

Peak Finder

Maximum number of peaks to find

Enter the number of extrema to look for in each input signal. The block stops searching the input signal for extrema once the maximum number of extrema has been found. The value of this parameter must be an integer greater than or equal to one.

Ignore peaks within threshold of neighboring values

Select this check box if you want to eliminate the detection of peaks whose amplitudes are within a specified threshold of neighboring values.

Threshold

Enter your threshold value. This parameter appears if you select the **Ignore peaks within threshold of neighboring values** check box.

When you select the **Ignore peaks within threshold of neighboring values** check box, the **Data Types** pane of the Peak Finder block appears as follows.

Function Block Parameters: Peak Finder

Peak Finder

Output the number of extrema (maxima and minima) in an input signal.

| Main | Data Types |

Settings on this pane only apply when block inputs are fixed-point signals.

Fixed-point operational parameters

Overflow mode: Wrap

OK   Cancel   Help   Apply
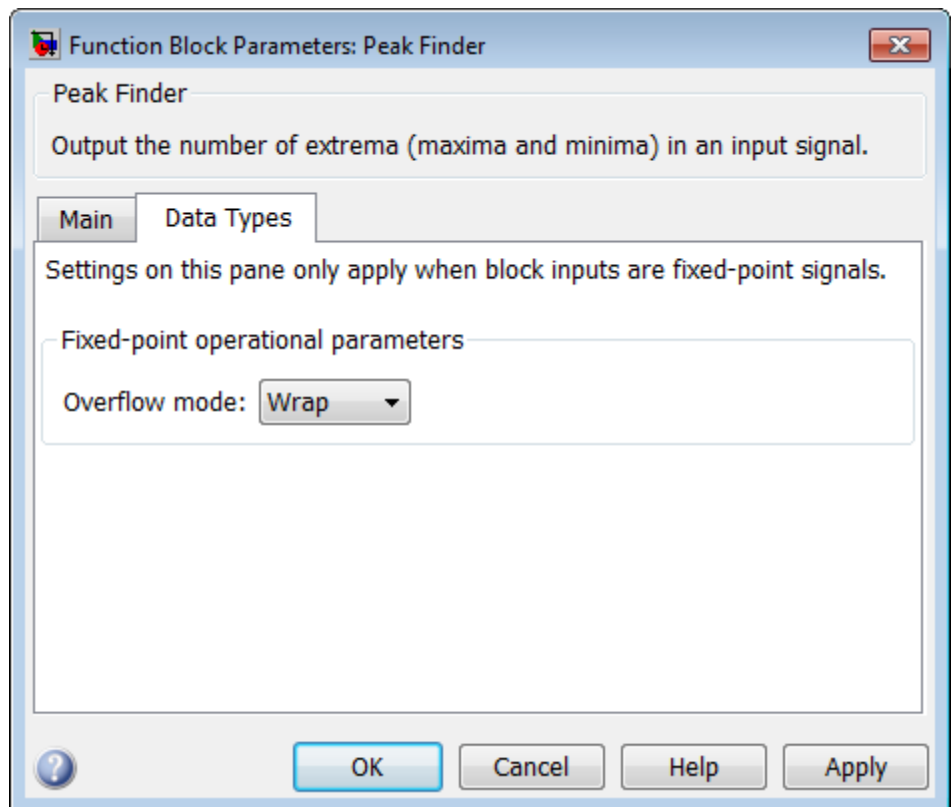
Overflow mode
> Select the overflow mode to be used when block inputs are fixed point.

Peak Finder

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Cnt	• 32-bit unsigned integers
Idx	• 32-bit unsigned integers
Val	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Pol	• Boolean

See Also

Maximum	DSP System Toolbox
Minimum	DSP System Toolbox

Purpose   Design peak or notch filter

Library   Filtering / Filter Designs

                     dspfdesign

Description   This block brings the filter design capabilities of the filterbuilder
                     function to the Simulink environment.
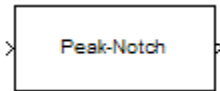
Dialog Box   See "Peak/Notch Filter Design Dialog Box — Main Pane" on page 7-767
                     for more information about the parameters of this block. The **Data
                     Types** and **Code Generation** panes are not available for blocks in the
                     DSP System Toolbox Filter Designs library.

                     Parameters of this block that do not change filter order or structure
                     are tunable.

Peak-Notch Filter

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this area you can specify whether you want to design a peaking filter or a notching filter, as well as the order of the filter.

Response

Select Peak or Notch from the drop-down list. The rest of the parameters that specify are equivalent for either filter type.

Order

Enter the filter order. The order must be even.

Frequency Specifications

This group of parameters allows you to specify frequency constraints and units.

Frequency Constraints

Select the frequency constraints for filter specification. There are two choices as follows:

- Center frequency and quality factor

- Center frequency and bandwidth

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, the block assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (O to 1), Hz, kHz, MHz, GHz.

Input Fs

This input box is enabled if **Frequency units** other than `Normalized (0 to 1)` are specified. Enter the input sampling frequency.

Center frequency

Enter the center frequency in the units specified previously.

Quality Factor

This input box is enabled only when `Center frequency and quality factor` is chosen for the **Frequency Constraints**. Enter the quality factor.

Bandwidth

This input box is enabled only when `Center frequency and bandwidth` is chosen for the **Frequency Constraints**. Enter the bandwidth.

Magnitude Specifications

This group of parameters allows you to specify the magnitude constraints, as well as their values and units.

Magnitude Constraints

Depending on the choice of constraints, the other input boxes are enabled or disabled. Select from four magnitude constraints available:

- `Unconstrained`

- `Passband ripple`

- `Stopband attenuation`

- `Passband ripple and stopband attenuation`

Magnitude units

Select the magnitude units: either `dB` or `squared`.

Apass

> This input box is enabled if the magnitude constraints selected
> are `Passband ripple` or `Passband ripple and stopband`
> `attenuation`. Enter the passband ripple.

Astop

> This input box is enabled if the magnitude constraints selected
> are `Stopband attenuation` or `Passband ripple and stopband`
> `attenuation`. Enter the stopband attenuation.

Algorithm

The parameters in this group allow you to specify the design method
and structure of your filter.

Design Method

> Lists all design methods available for the frequency and
> magnitude specifications you entered. When you change the
> specifications for a filter the methods available to design filters
> changes as well.

Scale SOS filter coefficients to reduce chance of overflow

> Selecting this parameter directs the design to scale the filter
> coefficients to reduce the chances that the inputs or calculations
> in the filter overflow and exceed the representable range of the
> filter. Clearing this option removes the scaling. This parameter
> applies only to IIR filters.

Filter Implementation

Structure

> Specify filter structure. Choose from:
>
> - Direct-form I SOS
>
> - Direct-form II SOS
>
> - Direct-form I transposed SOS
>
> - Direct-form II transposed SOS

Peak-Notch Filter

Use basic elements to enable filter customization
Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of n unit delays with a single delay by n.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values
Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing
Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

> **Note** The Inherited (this choice will be removed — see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- Enforce single-rate processing — When you select this option, the block maintains the sample rate of the input.

- Allow multirate processing — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to Elements as channels (sample based).

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Periodogram

Purpose

Power spectral density or mean-square spectrum estimate using periodogram method

Library

Estimation / Power Spectrum Estimation

`dspspect3`

Description
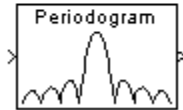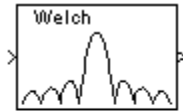
The Periodogram block estimates the power spectral density (PSD) or mean-square spectrum (MSS) of the input. It does so by using the periodogram method and Welch's averaged, modified periodogram method. The block averages the squared magnitude of the FFT computed over windowed sections of the input. It then normalizes the spectral average by the square of the sum of the window samples. See "Periodogram" and "Welch's Method" in the Signal Processing Toolbox documentation for more information.

The block treats M-by-N frame-based matrix input and M-by-N sample-based matrix input 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.

Each column of the output matrix contains the estimate of the power spectral density of the corresponding input column at N_{fft} equally spaced frequency points. The frequency points are in the range $[0, F_s)$, where F_s is the sampling frequency of the signal. The block always outputs sample-based data.
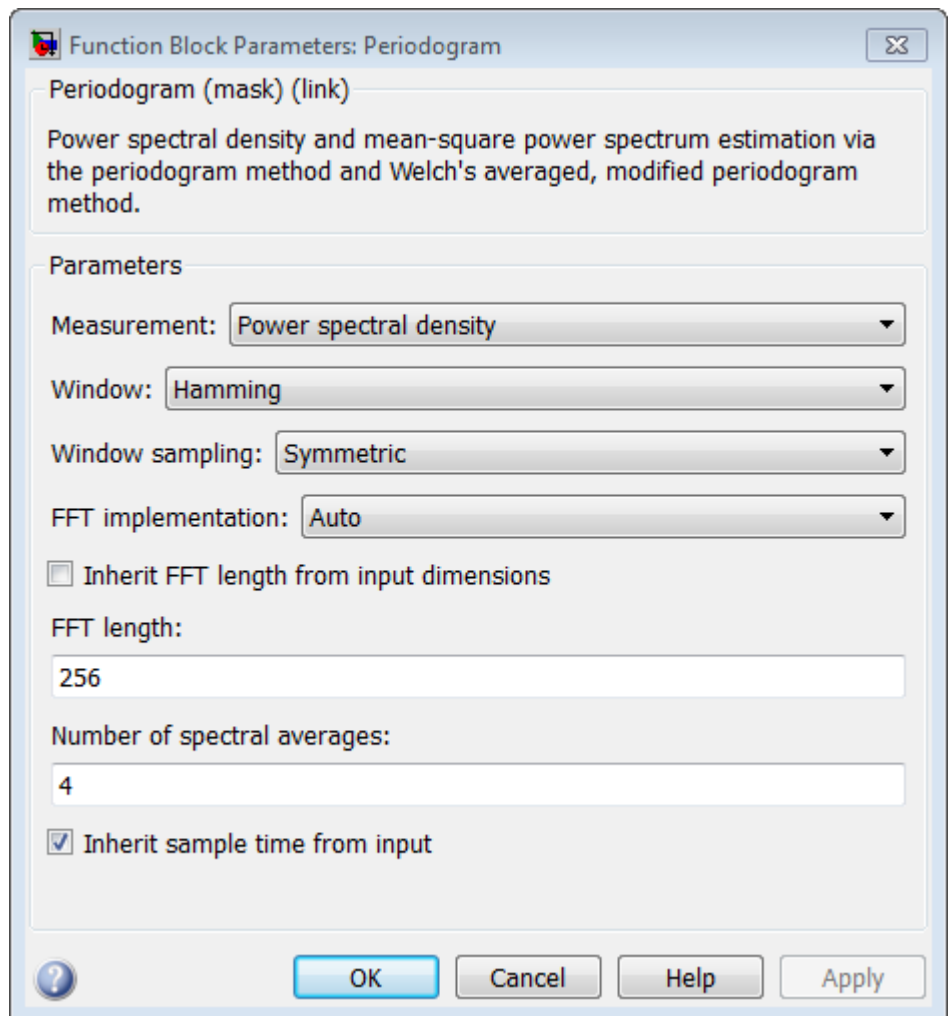
**Dialog
Box**

Measurement

Specify the type of measurement for the block to perform: `Power spectral density` or `Mean-square spectrum`. Tunable.

Window

> Select the type of window to apply. See the Window Function block reference page for more details. Tunable.

Stopband attenuation in dB

> Enter the level, in decibels (dB), of stopband attenuation, R_s, for the Chebyshev window. This parameter becomes visible if, for the **Window** parameter, you choose `Chebyshev`. Tunable.

Beta

> Enter the β parameter for the Kaiser window. This parameter becomes visible if, for the **Window** parameter, you chose `Kaiser`. Increasing **Beta** widens the mainlobe and decreases the amplitude of the sidelobes in the displayed frequency magnitude response. Tunable. See the Window Function block reference page for more details.

Window sampling

> From the list, choose `Symmetric` or `Periodic`. See the Window Function block reference page for more details.Tunable.

FFT implementation

> Set this parameter to `FFTW` [1], [2] to support an arbitrary length input signal. The block restricts generated code with FFTW implementation to MATLAB host computers.

> Set this parameter to `Radix-2` for bit-reversed processing, fixed or floating-point data, or for portable C-code generation using the Simulink Coder. The first dimension *M*, of the input matrix must be a power of two. To work with other input sizes, use the Pad block to pad or truncate these dimensions to powers of two, or if possible choose the FFTW implementation.

> Set this parameter to `Auto` to let the block choose the FFT implementation. For non-power-of-two transform lengths, the block restricts generated code to MATLAB host computers.

Inherit FFT length from input dimensions

When you select this check box, the block uses the input frame size as the number of data points, N_{fft}, on which to perform the FFT. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from estimation order** check box. You can then specify a power of two FFT length using the **FFT length** parameter.

FFT length

Enter the number of data points on which to perform the FFT, N_{fft}. When N_{fft} is larger than the input frame size, the block zero-pads each frame as needed. When N_{fft} is smaller than the input frame size, the block wraps each frame as needed. This parameter becomes visible only when you clear the **Inherit FFT length from input dimensions** check box.

When you set the **FFT implementation** parameter to Radix-2, this value must be a power of two.

Number of spectral averages

Specify the number of spectra to average. When you set this value to 1, the block computes the periodogram of the input. When you set this value greater 1, the block implements "Welch's Method" to compute a modified periodogram of the input.

Inherit sample time from input

If you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must 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 equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

Sample time of original time series
Specify the sample time of the original time-domain signal. This parameter becomes visible only when you clear the **Inherit sample time from input** check box.

Example   The dspstfft example provides an illustration of using the Periodogram and Matrix Viewer blocks to create a spectrogram. The dspsacomp example compares the Periodogram block with several other spectral estimation methods.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Orfanidis, S. J. *Introduction to Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

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

References

[1] FFTW (http://www.fftw.org)

[2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

See Also

Burg Method	DSP System Toolbox
Inverse Short-Time FFT	DSP System Toolbox
Magnitude FFT	DSP System Toolbox
Short-Time FFT	DSP System Toolbox
Spectrum Scope	DSP System Toolbox
Window Function	DSP System Toolbox
Yule-Walker Method	DSP System Toolbox
spectrum.periodogram	Signal Processing Toolbox
spectrum.welch	Signal Processing Toolbox

See "Spectral Analysis" for related information.

Permute Matrix

Purpose   Reorder matrix rows or columns

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

Description
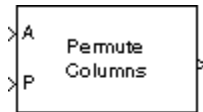
The Permute Matrix block reorders the rows or columns of M-by-N input matrix A as specified by indexing input P.

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, the block treats length-M unoriented vector input at the A port as an 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, the block treats length-N unoriented vector input at the A port as a 1-by-N matrix.

When an index value in input P references a nonexistent row or column of matrix A, the block reacts with the behavior specified by the **Invalid permutation index** parameter. The following options are available:

- Clip index — Clip the index to the nearest valid value (1 or M for row permutation, and 1 or N for column permutation), and *do not* issue an alert. Example: For a 3-by-7 input matrix, a column index of 9 is clipped to 7, and a row index of -2 is clipped to 1.

- `Clip and warn` — Display a warning message in the MATLAB command window, and clip the index as described above.

- `Generate error` — Display an error dialog box and terminate the simulation.

When length of the permutation vector `P` is not equal to the number of rows or columns of the input matrix `A`, you can choose to get an error dialog box and terminate the simulation by selecting **Error when length of P is not equal to Permute dimension size**.

Examples   In the model below, the top Permute Matrix block places the second row of the input matrix in both the first and fifth rows of the output matrix, and places the third row of the input matrix in the three middle rows of the output matrix. The bottom Permute Matrix block places the second column of the input matrix in both the first and fifth columns of the output matrix, and places the third column of the input matrix in the three middle columns of the output matrix.
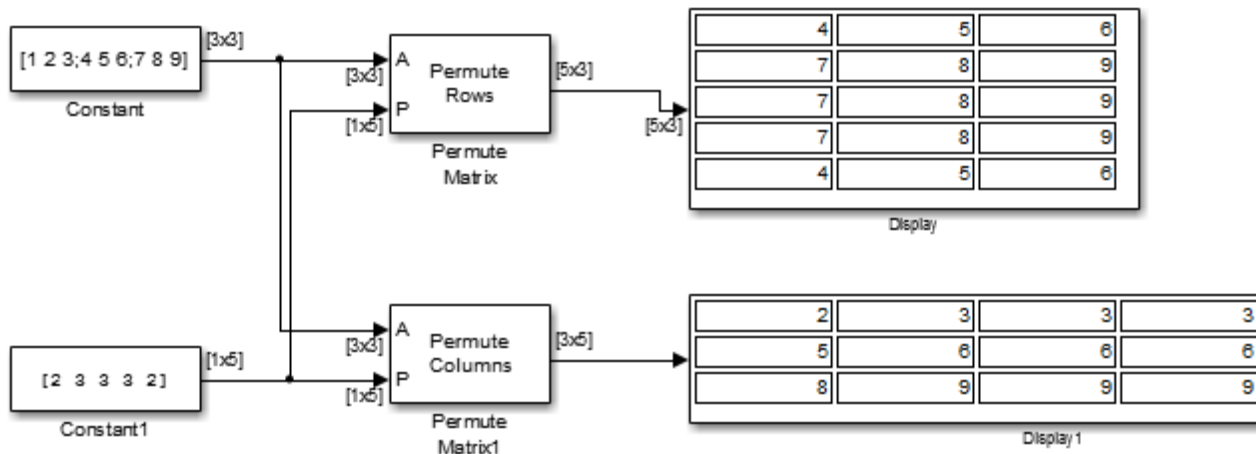
As shown in the example above, rows and columns of `A` can appear any number of times in the output, or not at all.

Permute Matrix
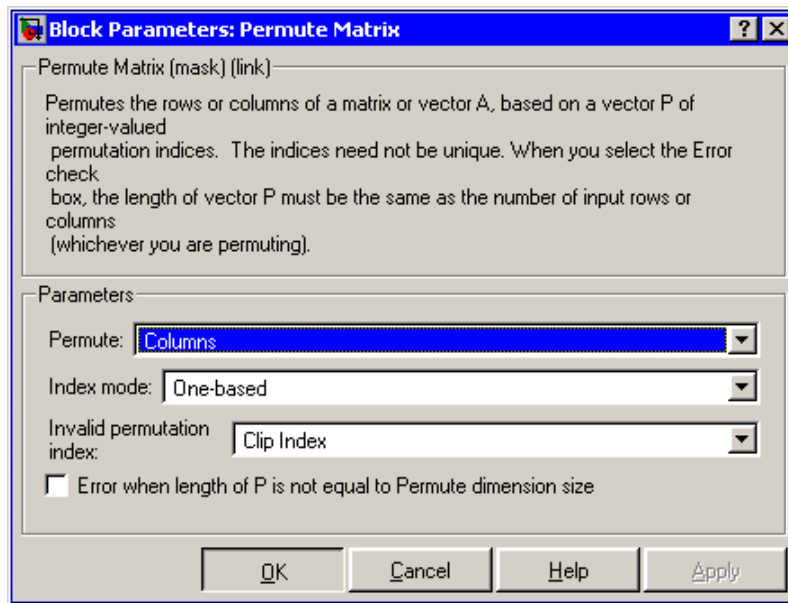
Dialog Box

Permute
> Method of constructing the output matrix; by permuting rows or columns of the input.

Index mode
> When set to One-based, a value of 1 in the permutation vector P refers to the first row or column of the input matrix A. When set to Zero-based, a value of 0 in P refers to the first row or column of A.

Invalid permutation index
> Response to an invalid index value. Tunable.

Error when length of P is not equal to Permute dimension size
> Option to display an error dialog box and terminate the simulation when the length of the permutation vector P is not equal to the number of rows or columns of the input matrix A.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	• Enumerated
P	• Double-precision floating point
	• Single-precision floating point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	• Enumerated
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	• Enumerated

See Also

Submatrix	DSP System Toolbox
Transpose	DSP System Toolbox

Permute Matrix

Variable Selector	DSP System Toolbox
permute	MATLAB

See "Reorder Channels in Multichannel Frame-Based Signals" for related information.

Purpose   Evaluate polynomial expression

Library   Math Functions / Polynomial Functions

dsppolyfun

Description   The Polynomial Evaluation block applies a polynomial function to the
real or complex input at the In port.
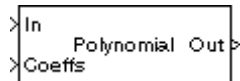
```
y = polyval(u)        % Equivalent MATLAB code
```

The Polynomial Evaluation block performs these types of operation
more efficiently than the equivalent construction using Simulink Sum
and Math Function blocks.

When you select the **Use constant coefficients** check box, you specify
the polynomial expression in the **Constant coefficients** parameter.
When you do not select **Use constant coefficients**, a variable
polynomial expression is specified by the input to the Coeffs port. In
both cases, the polynomial is specified as a vector of real or complex
coefficients in order of descending exponents.

The table below shows some examples of the block's operation for
various coefficient vectors.

Coefficient Vector	Equivalent Polynomial Expression
[1 2 3 4 5]	$y = u^4 + 2u^3 + 3u^2 + 4u + 5$
[1 0 3 0 5]	$y = u^4 + 3u^2 + 5$
[1 2+i 3 4-3i 5i]	$y = u^4 + (2+i)u^3 + 3u^2 + (4-3i)u + 5i$

Each element of a vector or matrix input to the In port is processed
independently, and the output size is the same as the input.

Polynomial Evaluation
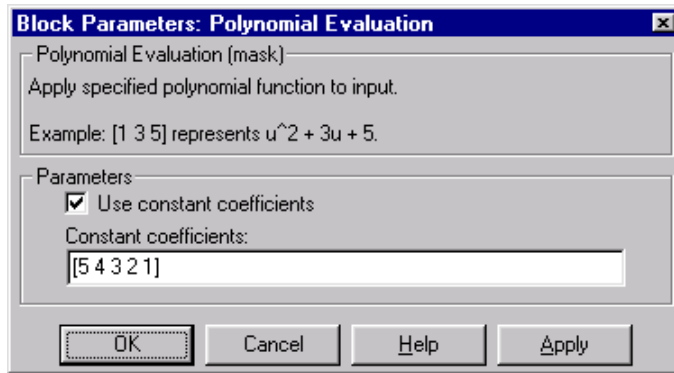
Dialog Box

Use constant coefficients
    Select to enable the **Constant coefficients** parameter and
    disable the Coeffs input port.

Constant coefficients
    Specify the vector of polynomial coefficients to apply to the input,
    in order of descending exponents. This parameter is enabled when
    you select the **Use constant coefficients** check box.

Supported Data Types

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

See Also

Least Squares Polynomial Fit	DSP System Toolbox
Math Function	Simulink
Sum	Simulink
polyval	MATLAB

Purpose   Use Schur-Cohn algorithm to determine whether all roots of input polynomial are inside unit circle

Library   Math Functions / Polynomial Functions

dsppolyfun
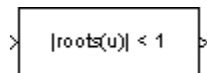
Description   The Polynomial Stability Test block uses the Schur-Cohn algorithm to determine whether all roots of a polynomial are within the unit circle.


```
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,

$$f(x) = u_1 x^{M-1} + u_2 x^{M-2} + \ldots + u_M$$

arranged in order of descending exponents, u_1, u_2, ..., u_M. The polynomial has order M-1 and positive integer exponents.

Inputs to the block represent the polynomial coefficients as shown in the previous equation. The block always treats length-M unoriented vector input as an M-by-1 matrix.

The output is a 1-by-N matrix with each column containing the value 1 or 0. The value 1 indicates that the polynomial in the corresponding column of the input is stable; that is, the magnitudes of all solutions to $f(x) = 0$ are less than 1. The value 0 indicates that the polynomial in the corresponding column of the input might be unstable; that is, the magnitude of at least one solution to $f(x) = 0$ is greater than or equal to 1.

Applications

This block is most commonly used to check the pole locations of the denominator polynomial, $A(z)$, of a transfer function, $H(z)$.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}}{a_1 + a_2 z^{-1} + \ldots + a_n z^{-(n-1)}}$$

Polynomial Stability Test

The poles are the n-1 roots of the denominator polynomial, $A(z)$. When any poles are located outside the unit circle, the transfer function $H(z)$ is unstable. As is typical in DSP applications, the transfer function above is specified in descending powers of z^{-1} rather than z.
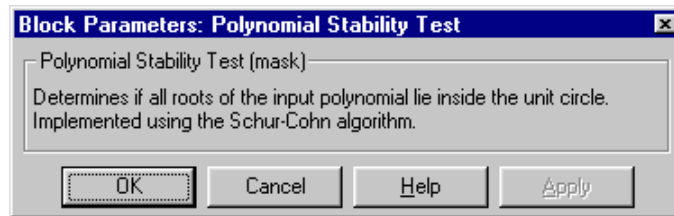
Dialog Box

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Boolean — Block outputs are always Boolean.

See Also

Least Squares Polynomial Fit	DSP System Toolbox
Polynomial Evaluation	DSP System Toolbox
polyfit	MATLAB

Purpose   Compute Moore-Penrose pseudoinverse of matrix

Library   Math Functions / Matrices and Linear Algebra / Matrix Inverses

dspinverses
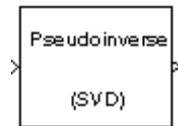
Description   The Pseudoinverse block computes the Moore-Penrose pseudoinverse of input matrix A.

```
[U,S,V] = svd(A,0)        % Equivalent MATLAB code
```

The pseudoinverse of A is the matrix $A^\dagger$ such that

$$A^\dagger = VS^\dagger U^*$$

where U and V are orthogonal matrices, and S is a diagonal matrix. The pseudoinverse has the following properties:

- $AA^\dagger = (AA^\dagger)^*$

- $A^\dagger A = (A^\dagger A)^*$

- $AA^\dagger A = A$

-

Pseudoinverse
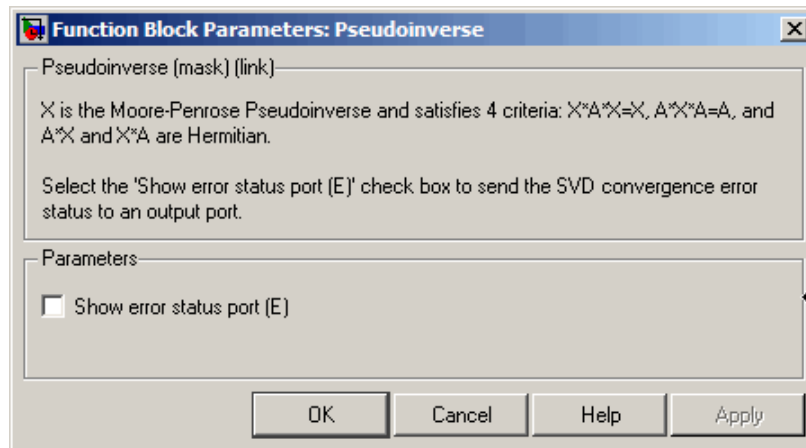
Dialog Box

Show error status port

Select to enable the E output port, which reports a failure to converge. The possible values you can receive on the port are:

- 0 — The pseudoinverse calculation converges.

- 1 — The pseudoinverse calculation does not converge.

If the pseudoinverse calculation fails to converge, the output at port X is an undefined matrix of the correct size.

References Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point
X	• Double-precision floating point • Single-precision floating point
E	• Boolean

See Also

Cholesky Inverse	DSP System Toolbox
LDL Inverse	DSP System Toolbox
LU Inverse	DSP System Toolbox
Singular Value Decomposition	DSP System Toolbox
inv	MATLAB

See "Matrix Inverses" for related information.

Pulse Shaping Filter

Purpose   Design pulse shaping filter

Library   Filtering / Filter Designs

                   dspfdesign

Description   This block brings the filter design capabilities of the `filterbuilder`
                   function to the Simulink environment.

**Dialog
Box**   See "Pulse-shaping Filter Design Dialog Box—Main Pane" on page
                   7-771 for more information about the parameters of this block. The
                   Data Types and **Code** panes are not available for blocks in the DSP
                   System Toolbox Filter Designs library.

                   Parameters of this block that do not change filter order or structure
                   are tunable.

Pulse Shaping Filter

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox product. You can use the tool to display:

- Magnitude response, phase response, and group delay in the frequency domain.

- Impulse response and step response in the time domain.

- Pole-zero information.

The tool also helps you evaluate filter performance by providing information about filter order, stability, and phase linearity. For more information on FVTool, see the Signal Processing Toolbox documentation.

Filter Specifications

In this group, you specify the shape and length of the filter.

Pulse shape

Select the shape of the impulse response from the following options:

- `Raised Cosine`

- `Square Root Raised Cosine`

- `Gaussian`

Order mode

This specification is only available for raised cosine and square root raised cosine filters. For these filters, select one of the following options:

- `Minimum`— This option will result in the minimum-length filter satisfying the user-specified **Frequency specifications**.

- `Specify order`—This option allows the user to construct a raised cosine or square root cosine filter of a specified order by entering an even number in the **Order** input box. The length of the impulse response will be `Order+1` .

- `Specify symbols`—This option enables the user to specify the length of the impulse response in an alternative manner. If `Specify symbols` is chosen, the **Order** input box changes to the **Number of symbols** input box.

Samples per symbol

Specify the oversampling factor. Increasing the oversampling factor guards against aliasing and improves the FIR filter approximation to the ideal frequency response. If **Order** is specified in **Number of symbols**, the filter length will be **Number of symbols*Samples per symbol**+1. The product **Number of symbols*Samples per symbol** must be an even number.

If a Gaussian filter is specified, the filter length must be specified in **Number of symbols** and **Samples per symbol**. The product **Number of symbols*Samples per symbol** must be an even number. The filter length will be **Number of symbols*Samples per symbol**+1.

Frequency specifications

In this group, you specify the frequency response of the filter. For raised cosine and square root raised cosine filters, the frequency specifications include:

Rolloff factor

The rolloff factor takes values in the range [0,1]. The smaller the rolloff factor, the steeper the transition in the stopband.

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, the block assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1)`, `Hz`, `kHz, MHz, GHz`

For a Gaussian pulse shape, the available frequency specifications are:

Bandwidth-time product
> This option allows the user to specify the width of the Gaussian filter. Note that this is independent of the length of the filter. The bandwidth-time product (BT) must be a positive real number. Smaller values of the bandwidth-time product result in larger pulse widths in time and steeper stopband transitions in the frequency response.

Frequency units
> The frequency units are normalized by default. If you specify units other than normalized, the block assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz

Magnitude specifications

If the **Order mode** is specified as minimum, the magnitude units may be selected from:

- dB — Specify the magnitude in decibels (default).

- Linear — Specify the magnitude in linear units.

Algorithm

The only design method available for FIR pulse-shaping filters is the window method.

Filter Implementation

Structure
> For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. FIR filters use direct-form structure.

Use basic elements to enable filter customization
> Select this check box to implement the filter as a subsystem of basic Simulink blocks. Clear the check box to implement the filter as a high-level subsystem. By default, this check box is cleared.

The high-level implementation provides better compatibility across various filter structures, especially filters that would contain algebraic loops when constructed using basic elements. On the other hand, using basic elements enables the following optimization parameters:

- **Optimize for zero gains** — Terminate chains that contain Gain blocks with a gain of zero.

- **Optimize for unit gains** — Remove Gain blocks that scale by a factor of one.

- **Optimize for delay chains** — Substitute delay chains made up of *n* unit delays with a single delay by *n*.

- **Optimize for negative gains** — Use subtraction in Sum blocks instead of negative gains in Gain blocks.

Optimize for unit-scale values

Select this check box to scale unit gains between sections in SOS filters. This parameter is available only for SOS filters.

Input processing

Specify how the block should process the input. The available options may vary depending on he settings of the **Filter Structure** and **Use basic elements for filter customization** parameters. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Pulse Shaping Filter

> **Note** The Inherited (this choice will be removed see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

When the **Filter type** parameter specifies a multirate filter, select the rate processing rule for the block from following options:

- Enforce single-rate processing — When you select this option, the block maintains the sample rate of the input.

- Allow multirate processing — When you select this option, the block adjusts the rate at the output to accommodate an increased or reduced number of samples. To select this option, you must set the **Input processing** parameter to Elements as channels (sample based).

Use symbolic names for coefficients

Select this check box to enable the specification of coefficients using MATLAB variables. The available coefficient names differ depending on the filter structure. Using symbolic names allows tuning of filter coefficients in generated code. By default, this check box is cleared.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

Purpose   Factor arbitrary matrix into unitary and upper triangular components

Library   Math Functions / Matrices and Linear Algebra / Matrix Factorizations

dspfactors

Description   The QR Factorization block uses a sequence of Householder transformations to triangularize the input matrix A. The block factors a column permutation of the M-by-N input matrix A as

$$A_e = QR$$

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.

$$\left| r_{i+1,j+1} \right| < \left| r_{i,j} \right| \qquad i = j$$

The size of matrices Q and R depends on the setting of the **Output size** parameter:

• When you select Economy for the output size, Q is an M-by-min(M,N) unitary matrix, and R is a min(M,N)-by-N upper-triangular matrix.

```
[Q R E] = qr(A,0)   % Equivalent MATLAB code
```

• When you select Full for the output size, Q is an M-by-M unitary matrix, and R is a M-by-N upper-triangular matrix.

```
[Q R E] = qr(A)      % Equivalent MATLAB code
```

The block treats length-M unoriented vector input as an M-by-1 matrix.

QR Factorization

QR factorization is an important tool for solving linear systems of equations because of good error propagation properties and the invertability of unitary matrices:

$$Q^{-1} = Q'$$

where Q' is the complex conjugate transpose of Q.

Unlike LU and Cholesky factorizations, the matrix A does not need to be square for QR factorization. However, QR factorization requires twice as many operations as LU Factorization (Gaussian elimination).

Examples   The **Output size** parameter of the QR factorization block has two settings: Economy and Full. When the M-by-N input matrix A has dimensions such that $M > N$, the dimensions of output matrices Q and R differ depending on the setting of the **Output size** parameter. If, however, the size of the input matrix A is such that $M \leq N$, output matrices Q and R have the same dimensions, regardless of whether the **Output size** is set to Economy or Full.

The input to the QR Factorization block in the following model is a 5-by-2 matrix A. When you change the setting of the **Output size** parameter from Economy to Full, the dimensions of the output given by the QR Factorization block also change.

1 Open the model by typing ex_qrfactorization_ref at the MATLAB command line.

2 Double-click the QR Factorization block, set the **Output size** parameter to Economy, and run the model.
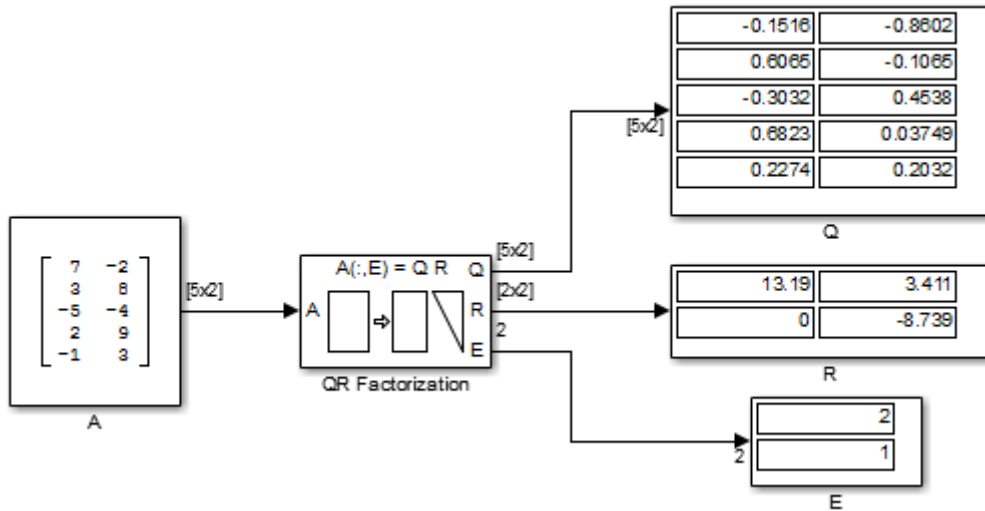
The QR Factorization block outputs a 5-by-2 matrix *Q* and a 2-by-2 matrix *R*.

3 Change the **Output size** parameter of the QR Factorization block to Full and rerun the model.

QR Factorization
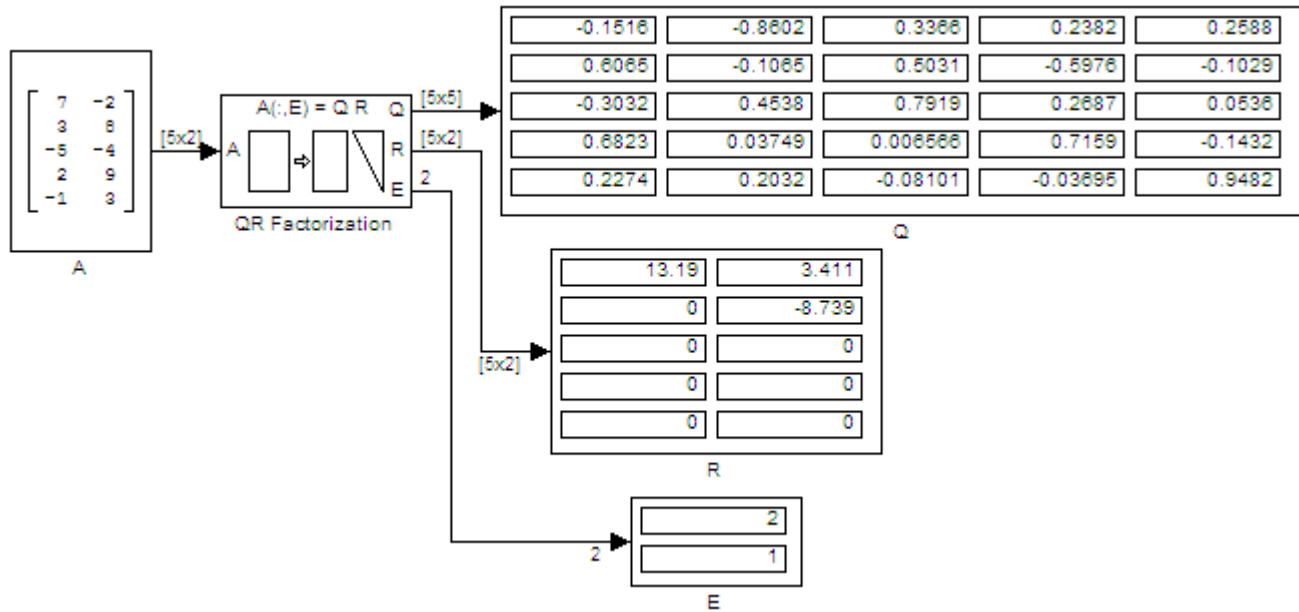
The QR Factorization block outputs a 5-by-5 matrix Q and a 5-by-2 matrix R.

Dialog Box

QR Factorization (mask) (link)

QR factorization with column pivoting. Q is a unitary matrix, R is an upper triangular matrix, and E is a permutation vector such that A(:,E) = Q*R.

Parameters

Output size: Economy

OK   Cancel   Help   Apply
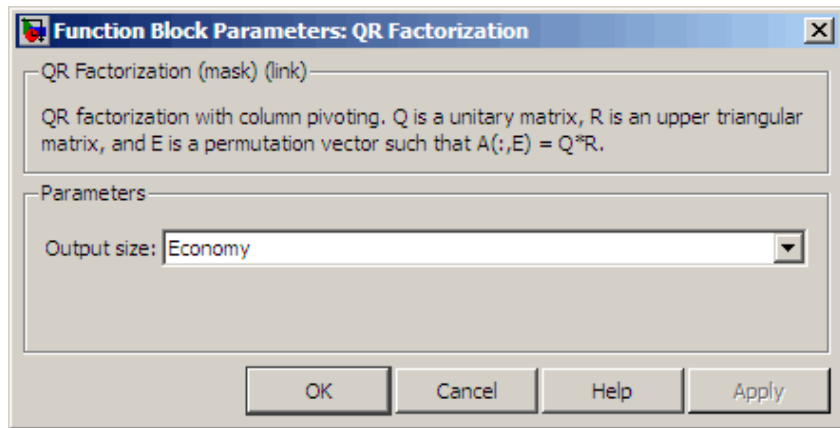
Output size

Specify the size of output matrices Q and R:

- Economy — When this output size is selected, the block outputs an M-by-min(M,N) unitary matrix Q and a min(M,N)-by-N upper-triangular matrix R.

- Full — When this output size is selected, the block outputs an M-by-M unitary matrix Q and a M-by-N upper-triangular matrix R.

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

QR Factorization

Purpose   Find minimum-norm-residual solution to A*X*=B

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers

dspsolvers
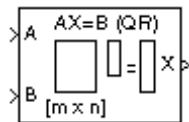
Description   The QR Solver block solves the linear system A*X*=B, which can be
                    overdetermined, underdetermined, or exactly determined. The system
                    is solved by applying QR factorization to the M-by-N matrix, A, at the
                    A port. The input to the B port is the right side M-by-L matrix, B. The
                    block treats length-M unoriented vector input as an M-by-1 matrix.

                    The output at the x port is the N-by-L matrix, X. X is chosen to minimize
                    the sum of the squares of the elements of B-AX. When B is a vector,
                    this solution minimizes the vector 2-norm of the residual (B-AX is the
                    residual). When B is a matrix, this solution minimizes the matrix
                    Frobenius norm of the residual. In this case, the columns of X are the
                    solutions to the L corresponding systems $AX_k=B_k$, where B_k is the kth
                    column of B, and X_k is the kth column of *X*.

                    X is known as the minimum-norm-residual solution to AX=B. The
                    minimum-norm-residual solution is unique for overdetermined
                    and exactly determined linear systems, but it is not unique for
                    underdetermined linear systems. Thus when the QR Solver is applied
                    to an underdetermined system, the output *X* is chosen such that the
                    number of nonzero entries in X is minimized.

Algorithm   QR factorization factors a column-permuted variant (A_e) of the M-by-N
                    input matrix A as

$$A_e = QR$$

where Q is a M-by-min(M,N) unitary matrix, and R is a min(M,N)-by-N
upper-triangular matrix.

The factored matrix is substituted for A_e in

$$A_e X = B_e$$

QR Solver

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.

$$RX = Y$$

─QR Solver (mask)─

Solve AX=B using QR factorization. B must have the same number of rows as A.

| OK | Cancel | Help | Apply |

Dialog Box

Supported Data Types

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

See Also

Levinson-Durbin	DSP System Toolbox
LDL Solver	DSP System Toolbox
LU Solver	DSP System Toolbox
QR Factorization	DSP System Toolbox
SVD Solver	DSP System Toolbox

See "Linear System Solvers" for related information.

Purpose   Discretize input at specified interval

Library   Quantizers

dspquant2

Description The Quantizer block is an implementation of the Simulink Quantizer block. See Quantizer for more information.

Queue

Purpose   Store inputs in FIFO register

Library   Signal Management / Buffers

dspbuff3

Description
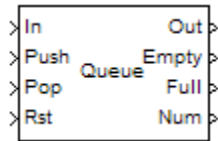
The Queue block stores a sequence of input samples in a first in, first out (FIFO) register. The register capacity is set by the **Register size** parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the In port onto the end of the queue when a trigger event is received at the Push port. When a trigger event is received at the Pop port, the block *pops* the first element off the queue and holds the Out port at that value. The first input to be pushed onto the queue is always the first to be popped off.
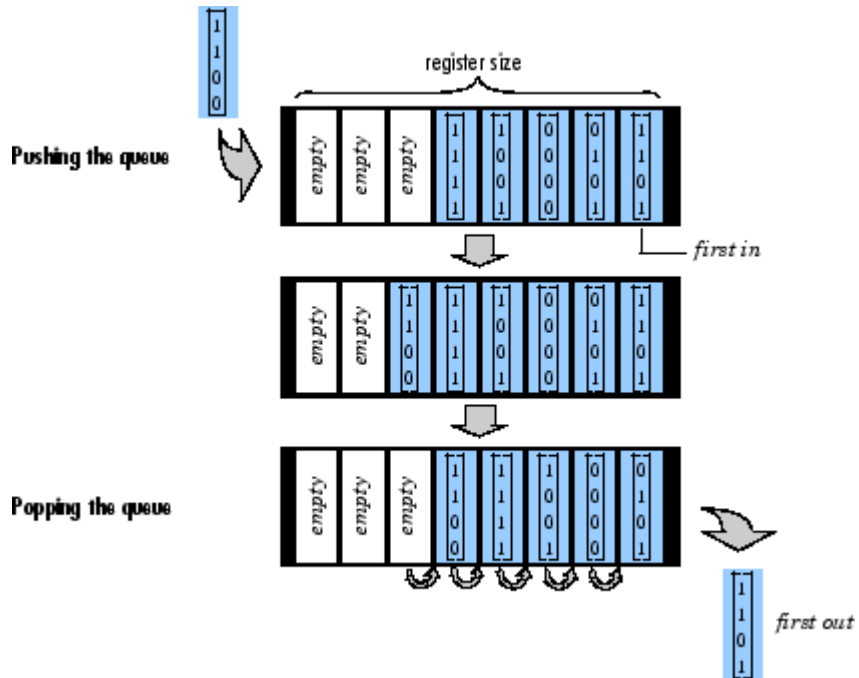
A trigger event at the optional Rst port empties the queue contents. When you select **Clear output port on reset**, then a trigger event at the Rst port empties the queue *and* sets the value at the Out port to zero. This setting also applies when a disabled subsystem containing the Queue block is reenabled; the Out port value is only reset to zero in this case when you select **Clear output port on reset**.

When you select the **Allow direct feedthrough** check box and two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

1 Rst

2 Push

3 Pop

When you clear the **Allow direct feedthrough** check box and two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

1 Rst

2 Pop

3 Push

The rate of the trigger signal must be the same as the rate of the data signal input. You specify the triggering event for the Push, Pop, and Rst ports by the **Trigger type** pop-up menu:

- Rising edge — Triggers execution of the block when the trigger input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero; see the following figure

- `Falling edge` — Triggers execution of the block when the trigger input does one of the following:

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

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

- `Either edge` — Triggers execution of the block when the trigger input is a `Rising edge` or `Falling edge` (as described above).

- `Non-zero sample` — Triggers execution of the block at each sample time that the trigger input is not zero.

Note If your model contains any referenced models that use a Queue
block with the **Push onto full register** parameter set to Dynamic
reallocation, you cannot simulate your top-level model in Simulink
Accelerator mode.

The **Push onto full register** parameter specifies the block's behavior
when a trigger is received at the Push port but the register is full. The
Pop empty register parameter specifies the block's behavior when
a trigger is received at the Pop port but the register is empty. The
following options are available for both cases:

- Ignore — Ignore the trigger event, and continue the simulation.
- Warning — Ignore the trigger event, but display a warning message
  in the MATLAB Command Window.
- Error — Display an error dialog box and terminate the simulation.

Note The **Push onto full register** and **Pop empty register**
parameters are diagnostic parameters. Like all diagnostic parameters
on the Configuration Parameters dialog box, they are set to Ignore in
the code generated for this block by Simulink Coder code generation
software.

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 **Show number of register entries port**
parameter.

Queue
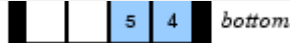
Examples   **Example 1**

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 `Rst` 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	Rst	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

In	Push	Pop	Rst	Queue	Out	Empty	Full	Num
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 Rst ports are triggered simultaneously. The Rst trigger takes precedence, and the queue is first cleared and then pushed.

Example 2

The dspqdemo example provides another example of the operation of the Queue block.

Queue
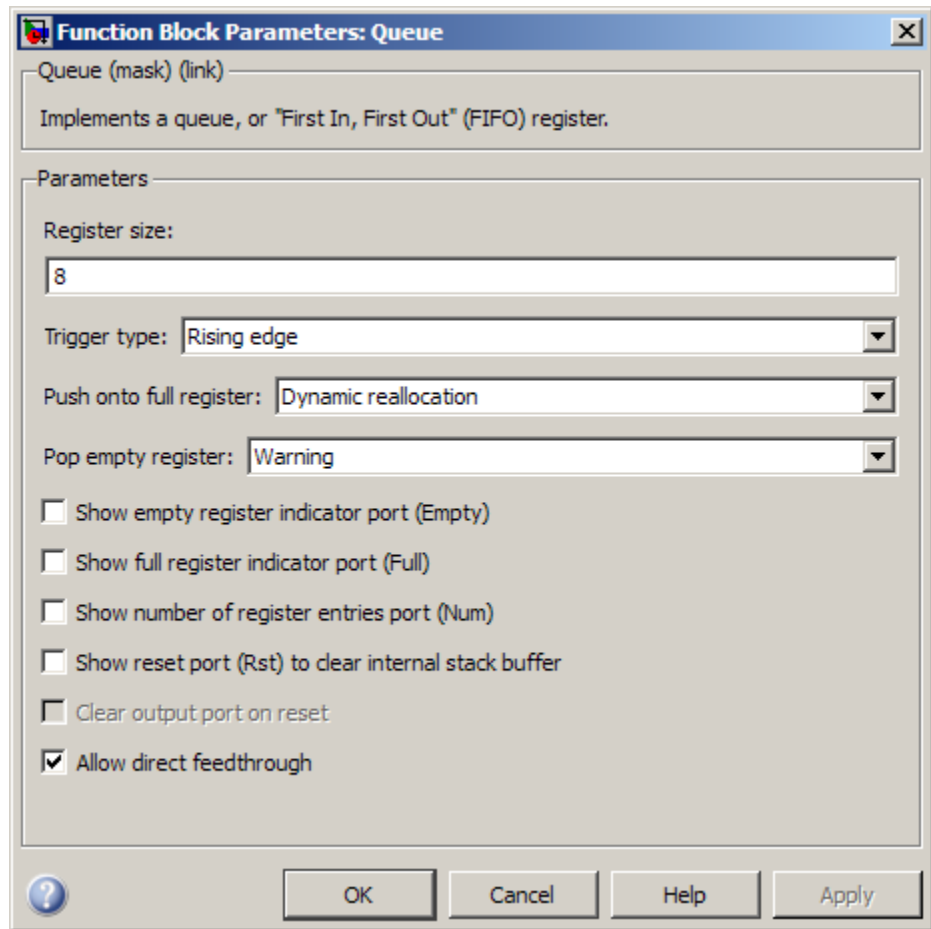
Register size
    The number of entries that the FIFO register can hold.

Trigger type
    The type of event that triggers the block's execution. The rate of
    the trigger signal must be the same as the rate of the data signal
    input.

Push onto full register

Response to a trigger received at the Push port when the register is full. Inputs to this port must have the same built-in data type as inputs to the Pop and Rst input ports.

When Dynamic reallocation is selected, the **System target file** parameter on the **Code Generation** pane of the Model Configuration Parameters dialog box must be set to grt_malloc.tlc   Generic Real-Time Target with dynamic memory allocation.

Pop empty register

Response to a trigger received at the Pop port when the register is empty. Inputs to this port must have the same built-in data type as inputs to the Push and Rst input ports.

Show empty register indicator port

Enable the Empty output port, which is high (1) when the queue is empty, and low (0) otherwise.

Show full register indicator port

Enable the Full output port, which is high (1) when the queue is full, and low (0) otherwise. The Full port remains low when you select Dynamic reallocation from the **Push onto full register** parameter.

Show number of register entries port

Enable the Num output port, which tracks the number of entries currently on the queue. When inputs to the In port are double-precision values, the outputs from the Num port are double-precision values. Otherwise, the outputs from the Num port are 32-bit unsigned integer values.

Show reset port to clear internal stack buffer

Enable the Rst input port, which empties the queue when the trigger specified by the **Trigger type** is received. Inputs to this port must have the same built-in data type as inputs to the Push and Pop input ports.

Clear output port on reset

Reset the Out port to zero, in addition to clearing the queue, when a trigger is received at the Rst input port.

Allow direct feedthrough

When you select this check box, the input data is available immediately at the output port of the block. You can turn off direct feedthrough and delay the input data by an extra frame by clearing the **Allow direct feedthrough** check box.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Push	• Double-precision floating point
	• Single-precision floating point
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	Inputs to this port must have the same built-in data type as inputs to the Pop and Rst input ports

Port	Supported Data Types
Pop	• Double-precision floating point
	• Single-precision floating point
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	Inputs to this port must have the same built-in data type as inputs to the Push and Rst input ports.
Rst	• Double-precision floating point
	• Single-precision floating point
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	Inputs to this port must have the same built-in data type as inputs to the Push and Pop input ports.
Out	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Empty	• Double-precision floating point
	• Boolean

Queue

Port	Supported Data Types
Full	• Double-precision floating point • Boolean
Num	• Double-precision floating point The block outputs a double-precision floating-point value at this port when the data type of the In port is double-precision floating-point. • 32-bit unsigned integers The block outputs a 32-bit unsigned integer value at this port when the data type of the In port is anything other than double-precision floating-point.

See Also

Buffer	DSP System Toolbox
Delay Line	DSP System Toolbox
Stack	DSP System Toolbox

Purpose   Generate randomly distributed values

Library   Sources

dspsrcs4

Description   The Random Source block generates a frame of M values drawn from a
uniform or Gaussian pseudorandom distribution, where you specify M
in the **Samples per frame** parameter.

This reference page contains a detailed discussion of the following
Random Source block topics:

- "Distribution Type" on page 2-1253
- "Output Complexity" on page 2-1254
- "Output Repeatability" on page 2-1256
- "Specifying the Initial Seed" on page 2-1256
- "Sample Period" on page 2-1257
- "Dialog Box" on page 2-1258
- "Supported Data Types" on page 2-1261
- "See Also" on page 2-1261

Distribution Type

When the **Source type** parameter is set to Uniform, the output samples
are drawn from a uniform distribution whose minimum and maximum
values are specified by the **Minimum** and **Maximum** parameters,
respectively. All values in this range are equally likely to be selected. A
length-N vector specified for one or both of these parameters generates
an N-channel output (M-by-N matrix) containing a unique random
distribution in each channel.

For example, specify

- **Minimum** = [0 0 -3 -3]

- **Maximum** = `[10 10 20 20]`

to generate a four-channel output whose first and second columns contain random values in the range [0, 10], and whose third and fourth columns contain random values in the range [-3, 20]. When you specify only one of the **Minimum** and **Maximum** parameters as a vector, the block scalar expands the other parameter so it is the same length as the vector.

When the **Source type** parameter is set to `Gaussian`, you must also set the **Method** parameter, which determines the method by which the block computes the output, and has the following settings:

- `Ziggurat` — Produces Gaussian random values by using the Ziggurat method.

- `Sum of uniform values` — Produces Gaussian random values by adding and scaling uniformly distributed random signals based on the central limit theorem. This theorem states that the probability distribution of the sum of a sufficiently high number of random variables approaches the Gaussian distribution. You must set the **Number of uniform values to sum** parameter, which determines the number of uniformly distributed random numbers to sum to produce a single Gaussian random value.

For both settings of the **Method** parameter, the output samples are drawn from the normal distribution defined by the **Mean** and **Variance** parameters. A length-N vector specified for one or both of the **Mean** and **Variance** parameters generates an N-channel output (M-by-N frame matrix) containing a distinct random distribution in each column. When you specify only one of these parameters as a vector, the block scalar expands the other parameter so it is the same length as the vector.

Output Complexity

The block's output can be either real or complex, as determined by the `Real` and `Complex` options in the **Complexity** parameter. These settings control all channels of the output, so real and complex data cannot be combined in the same output. For complex output with a

Uniform distribution, the real and imaginary components in each channel are both drawn from the same uniform random distribution, defined by the **Minimum** and **Maximum** parameters for that channel.

For complex output with a Gaussian distribution, the real and imaginary components in each channel are drawn from normal distributions with different means. In this case, the **Mean** parameter for each channel should specify a complex value; the real component of the **Mean** parameter specifies the mean of the real components in the channel, while the imaginary component specifies the mean of the imaginary components in the channel. When either the real or imaginary component is omitted from the **Mean** parameter, a default value of 0 is used for the mean of that component.

For example, a **Mean** parameter setting of [5+2i 0.5 3i] generates a three-channel output with the following means.

Channel 1 mean	*real* = 5	*imaginary* = 2
Channel 2 mean	*real* = 0.5	*imaginary* = 0
Channel 3 mean	*real* = 0	*imaginary* = 3

For complex output, the **Variance** parameter, σ^2, specifies the *total variance* for each output channel. This is the sum of the variances of the real and imaginary components in that channel.

$$\sigma^2 = \sigma_{Re}^2 + \sigma_{Im}^2$$

The specified variance is equally divided between the real and imaginary components, so that

$$\sigma_{Re}^2 = \frac{\sigma^2}{2}$$
$$\sigma_{Im}^2 = \frac{\sigma^2}{2}$$

Random Source

Output Repeatability

The **Repeatability** parameter determines whether or not the block outputs the same signal each time you run the simulation. You can set the parameter to one of the following options:

- `Repeatable` — Outputs the same signal each time you run the simulation. The first time you run the simulation, the block randomly selects an initial seed. The block reuses these same initial seeds every time you rerun the simulation.

- `Specify seed` — Outputs the same signal each time you run the simulation. Every time you run the simulation, the block uses the initial seed(s) specified in the **Initial seed** parameter. Also see "Specifying the Initial Seed" on page 2-1256.

- `Not repeatable` — Does not output the same signal each time you run the simulation. Every time you run the simulation, the block randomly selects an initial seed.

Specifying the Initial Seed

When you set the **Repeatability** parameter to `Specify seed`, you must set the **Initial seed** parameter. The **Initial seed** parameter specifies the initial seed for the pseudorandom number generator. The generator produces an identical sequence of pseudorandom numbers each time it is executed with a particular initial seed.

Specifying Initial Seeds for Real Outputs

To specify the N initial seeds for an N-channel real-valued output, **Complexity** parameter set to `Real`, provide one of the following in the **Initial seed** parameter:

- Length-N vector of initial seeds — Uses each vector element as an initial seed for the corresponding channel in the N-channel output.

- Single scalar — Uses the scalar to generate N random values, which it uses as the seeds for the N-channel output.

Specifying Initial Seeds for Complex Outputs

To specify the initial seeds for an N-channel complex-valued output, **Complexity** parameter set to Complex, provide one of the following in the **Initial seed** parameter:

- Length-N vector of initial seeds — Uses each vector element as an initial seed for generating N channels of *real* random values. The block uses pairs of adjacent values in each of these channels as the real and imaginary components of the final output, as illustrated in the following figure.

- Single scalar — Uses the scalar to generate N random values, which it uses as the seeds for generating N channels of *real* random values. The block uses pairs of adjacent values in each of these channels as the real and imaginary components of the final output, as illustrated in the following figure.
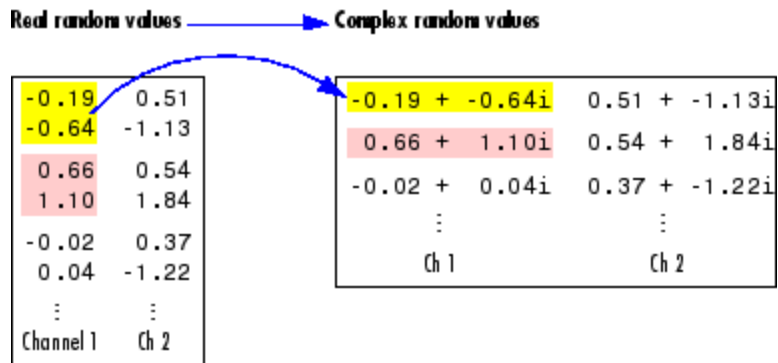
Sample Period

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

Random Source

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 DSP System Toolbox blocks do not accept continuous-time inputs.

Dialog Box

Only some of the parameters described below are visible in the dialog box at any one time.
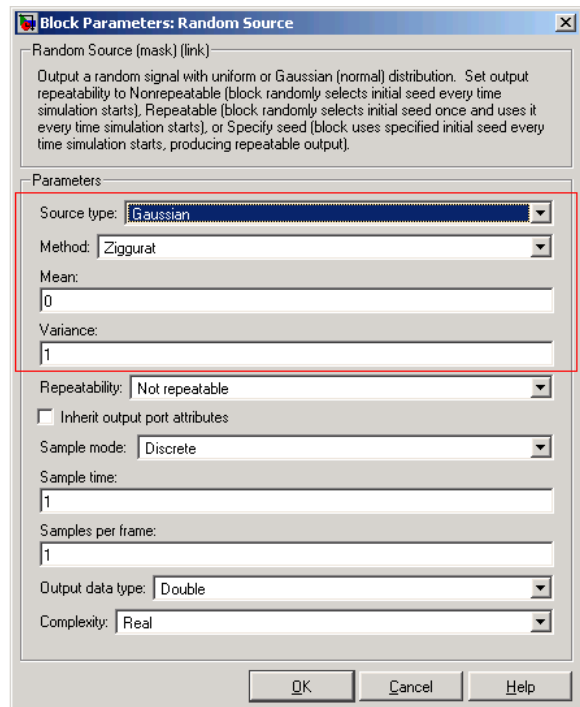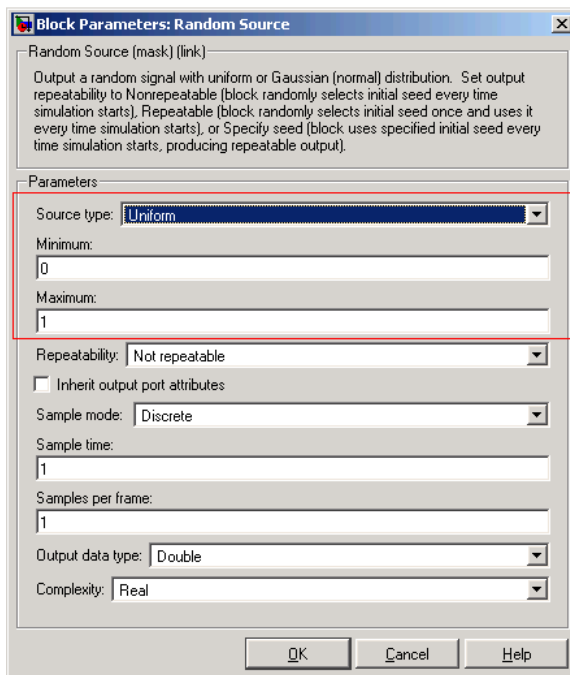
Source type

    The distribution from which to draw the random values, `Uniform` or `Gaussian`. For more information, see "Distribution Type" on page 2-1253.

Method

    The method by which the block computes the Gaussian random values, `Ziggurat` or `Sum of uniform values`. This parameter is enabled when **Source type** is set to `Gaussian`. For more information, see "Distribution Type" on page 2-1253.

Minimum

    The minimum value in the uniform distribution. This parameter is enabled when you select `Uniform` from the **Source type** parameter. Tunable.

Maximum

    The maximum value in the uniform distribution. This parameter is enabled when you select you select `Uniform` from the **Source type** parameter. Tunable.

Number of uniform values to sum

    The number of uniformly distributed random values to sum to compute a single number in a Gaussian random distribution. This parameter is enabled when the **Source type** parameter is set to `Gaussian`, and the **Method** parameter is set to `Sum of uniform values`. For more information, see "Distribution Type" on page 2-1253.

Mean

    The mean of the Gaussian (normal) distribution. This parameter is enabled when you select `Gaussian` from the **Source type** parameter. Tunable.

Variance

    The variance of the Gaussian (normal) distribution. This parameter is enabled when you select `Gaussian` from the **Source type** parameter. Tunable.

Random Source

Repeatability

The repeatability of the block output: `Not repeatable`, `Repeatable`, or `Specify seed`. In the `Repeatable` and `Specify seed` settings, the block outputs the same signal every time you run the simulation. For details, see "Output Repeatability" on page 2-1256.

Initial seed

The initial seed(s) to use for the random number generator when you set the **Repeatability** parameter to `Specify seed`. For details, see "Specifying the Initial Seed" on page 2-1256. Tunable.

Inherit output port attributes

When you select this check box, block inherits the sample mode, sample time, output data type, complexity, and signal dimensions of a sample-based signal from a downstream block. When you select this check box, the **Sample mode**, **Sample time**, **Samples per frame**, **Output data type**, and **Complexity** parameters are disabled.

Suppose you want to back propagate a 1-D vector. The output of the Random Source block is a length M sample-based 1-D vector, where length M is inherited from the downstream block. When the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter specifies N channels, the 1-D vector output contains M/N samples from each channel. An error occurs in this case when M is not an integer multiple of N.

Suppose you want to back propagate a M-by-N signal. When N>1, your signal has N channels. When N = 1, your signal has M channels. The value of the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter can be a scalar or a vector of length equal to the number of channels. You can specify these parameters as either row or column vectors, except when the signal is a row vector. In this case, the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter must also be specified as a row vector.

Sample mode

The sample mode, Continuous or Discrete. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Sample time

The sample period, T_s, of the random output sequence. The output frame period is $M*T_s$. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Samples per frame

The number of samples, M, in each output frame. When the value of this parameter is 1, the block outputs a sample-based signal.

This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Output data type

The data type of the output, single-precision or double-precision. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Complexity

The complexity of the output, Real or Complex. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Supported Data Types

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

See Also

Discrete Impulse	DSP System Toolbox
Maximum	DSP System Toolbox
Minimum	DSP System Toolbox

Random Source

Signal From Workspace	DSP System Toolbox
Standard Deviation	DSP System Toolbox
Variance	DSP System Toolbox
Constant	Simulink
Random Number	Simulink
Signal Generator	Simulink
rand	MATLAB
randn	MATLAB
RandStream	MATLAB

Purpose   Compute real cepstrum of input

Library   Transforms

dspxfrm3
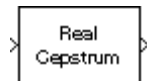
Description   The Real Cepstrum block computes the real cepstrum of each column in the real-valued M-by-N input matrix, u. The block treats each column of the input as an independent channel containing M consecutive samples. The block does not accept complex-valued inputs.

The output is a real M_o-by-N matrix, where you specify M_o in the **FFT length** parameter. Each output column contains the length-M_o real cepstrum of the corresponding input column.

```
y = real(ifft(log(abs(fft(u,Mo)))))
```

or, more compactly,

```
y = rceps(u,Mo)
```

When you select the **Inherit FFT length from input port dimensions** check box, the output frame size matches the input frame size ($M_o=M$). In this case, the block processes *sample-based* length-M row vector inputs as a single channel (that is, as an M-by-1 column vector), and returns the result as a length-M column vector. The block *always* processes unoriented vector inputs as a single channel, and returns the result as a length-M column vector.

The output port rate is the same as the input port rate.

Real Cepstrum
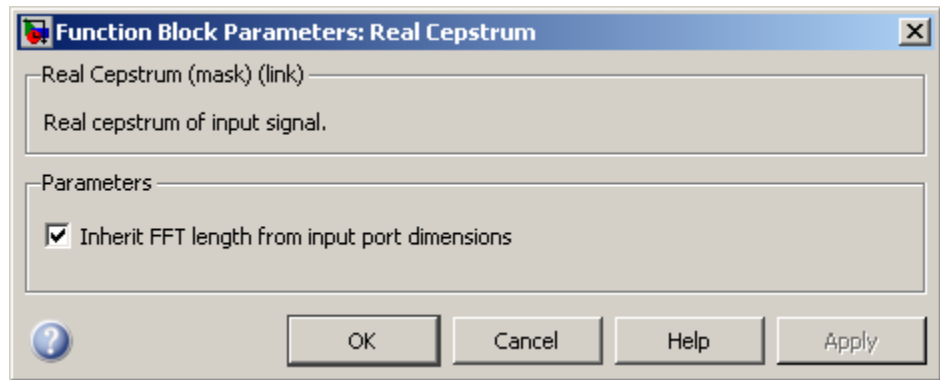
Dialog Box

Inherit FFT length from input port dimensions
When you select this check box, the output frame size matches 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 visible only when you clear the **Inherit FFT length from input port dimensions** check box.

Supported Data Types

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

See Also

Complex Cepstrum	DSP System Toolbox
DCT	DSP System Toolbox
FFT	DSP System Toolbox
rceps	Signal Processing Toolbox

Purpose   Compute reciprocal condition of square matrix in 1-norm

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

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 (κ ≥ 1), and *y* is the scalar output (0 ≤ *y* < 1).

The matrix 1-norm, $\|A\|_1$, is the maximum column-sum in the M-by-M matrix A.

$$\|A\|_1 = 1 \leq \overset{\max}{j} \leq M \sum_{i=1}^{M} \left| a_{ij} \right|$$

For a 3-by-3 matrix:

Reciprocal Condition
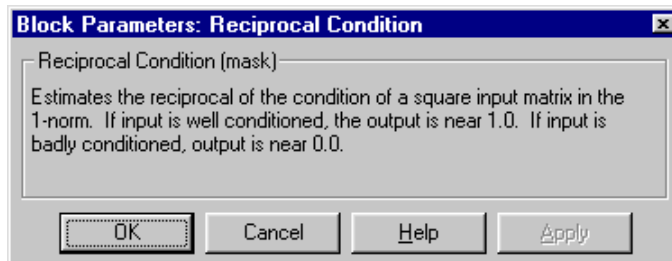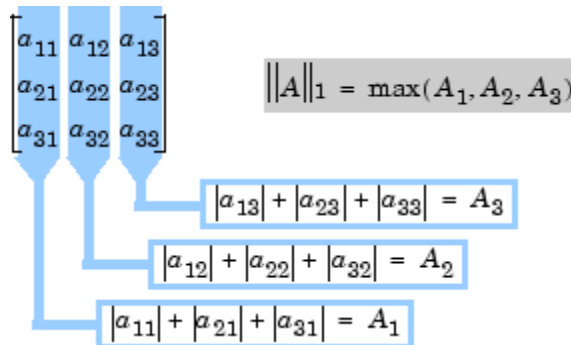
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad \|A\|_1 = \max(A_1, A_2, A_3)$$

$$|a_{13}| + |a_{23}| + |a_{33}| = A_3$$

$$|a_{12}| + |a_{22}| + |a_{32}| = A_2$$

$$|a_{11}| + |a_{21}| + |a_{31}| = A_1$$

Block Parameters: Reciprocal Condition

Reciprocal Condition (mask)

Estimates the reciprocal of the condition of a square input matrix in the 1-norm. If input is well conditioned, the output is near 1.0. If input is badly conditioned, output is near 0.0.

OK   Cancel   Help   Apply

Dialog Box

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types
- Double-precision floating point
- Single-precision floating point

See Also

Matrix 1-Norm	DSP System Toolbox
Normalization	DSP System Toolbox
rcond	MATLAB

Purpose   Design and apply equiripple FIR filter

Library   dspobslib

Description

Note The Remez FIR Filter Design block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Remez FIR Filter Design block implements the Parks-McClellan algorithm to design and apply a linear-phase filter with an arbitrary multiband magnitude response. The filter design, which uses the Signal Processing Toolbox firpm function, 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 = $[0 \quad 0.3 \quad 0.4 \quad 0.6 \quad 0.7 \quad 1]$

Gains = $[0 \quad 1 \quad 0 \quad 0 \quad 0.5 \quad 0.5]$

Band: ¿ ¡ ¬
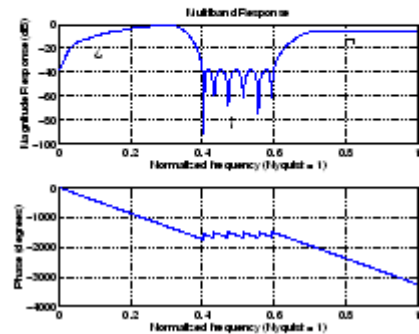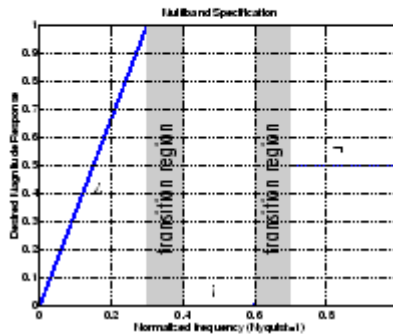
The **Weights** parameter is a vector that specifies the emphasis to be placed on minimizing the error in certain frequency bands relative to others. This vector specifies one weight per band, so it is half the length of the **Band-edge frequency vector** and **Gains at these frequencies** vectors.

In most cases, differentiators and Hilbert transformers have only a single band, so the weight is a scalar value that does not affect the final filter. However, the **Weights** parameter is useful when using the block to design an antisymmetric multiband filter, such as a Hilbert transformer with stopbands.

Examples

Example 1: Multiband

Consider a lowpass filter with a transition band in the normalized frequency range 0.4 to 0.5, and 10 times greater error minimization in the stopband than in the passband.

In this case:

- **Filter type** = Multiband

Remez FIR Filter Design (Obsolete)

- **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**
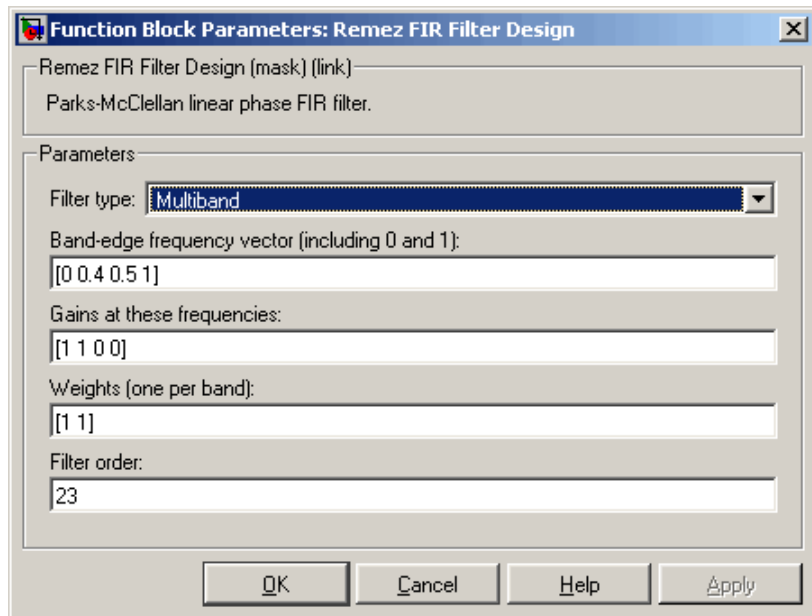
Filter type

    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.

Purpose   Resample input at higher rate by repeating values

Library   Signal Operations

dspsigops
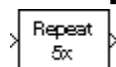
Description   The Repeat block upsamples each channel of the M_i-by-N input to a rate
L times higher than the input sample rate. To do so, the block repeats
each consecutive input sample L times at the output. You specify the
integer L in the **Repetition count** parameter.

You can use the Repeat block inside of triggered subsystems when you
set the **Rate options** parameter to Enforce single-rate processing.

Frame-Based Processing

When you set the **Input processing** parameter to Columns as
channels (frame based), the block upsamples each column of the
input over time. In this mode, the block can perform either single-rate
or multirate processing. You can use the **Rate options** parameter to
specify how the block upsamples the input:

- When you set the **Rate options** parameter to Enforce single-rate
  processing, the input and output of the block have the same sample
  rate. In this mode, the block outputs a signal with a proportionally
  larger frame *size* than the input. The block upsamples each channel
  independently by repeating each row of the input matrix L times at
  the output. 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.

  For an example of single-rate upsampling, see Example: Single-Rate
  Processing on page 2-1275.

- When you set the **Rate options** parameter to Allow multirate
  processing, the block treats an M_i-by-N matrix input as N
  independent channels. 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 input frame

period ($T_{fo} = T_{fi}/L$). In this mode, the output always has the same frame size as the input.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an *M*-by-*N* matrix input as *M**N independent channels, and upsamples each channel over time. The block upsamples each channel over time such that the output sample rate is *L* times higher than the input sample rate ($T_{so} = T_{si}/L$). In this mode, the output is always the same size as the input.

Zero Latency

The Repeat block has *zero-tasking latency* for all single-rate operations. The block is in a single-rate mode if you set the **Repetition count** parameter to `1` or if you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`.

The Repeat block also has zero-tasking latency for multirate operations if you run your model in Simulink single-tasking mode.

Zero-tasking latency means that the block repeats the first input (received at *t*=0) for the first *L* output samples, the second input for the next *L* output samples, and so on.

Nonzero Latency

The Repeat block has tasking latency for multirate, multitasking operation:

- In multirate, sample-based processing 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 conditions** 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 multirate, frame-based processing 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 conditions** parameter can be an M_i-by-*N* matrix, or a scalar to be repeated across all elements of the M_i-by-*N* matrix.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

Examples   ### Example: Single-Rate Processing

In the ex_repeat_ref2 model, the Repeat block resamples a single-channel input with a frame size of 16. The block repeats input values to upsample the input by a factor of 4. Thus, the output of the block has a frame size of 64. The input and output frame rates are identical.
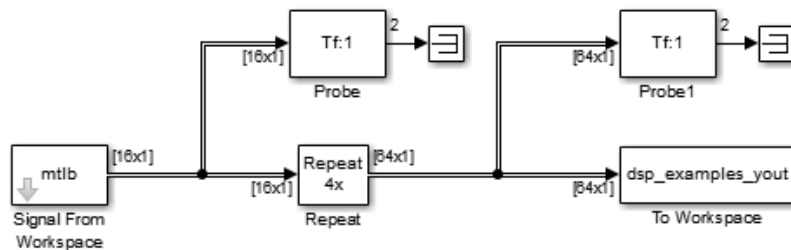
Example: Multirate, Frame-Based Processing

In the ex_repeat_ref1 model, the Repeat block resamples a single-channel input with a frame period of 1 second. The block repeats input values to upsample the input by a factor of 4. Thus, the output of the block has a frame period of 0.25 seconds. The input and output frame sizes are identical.

Repeat
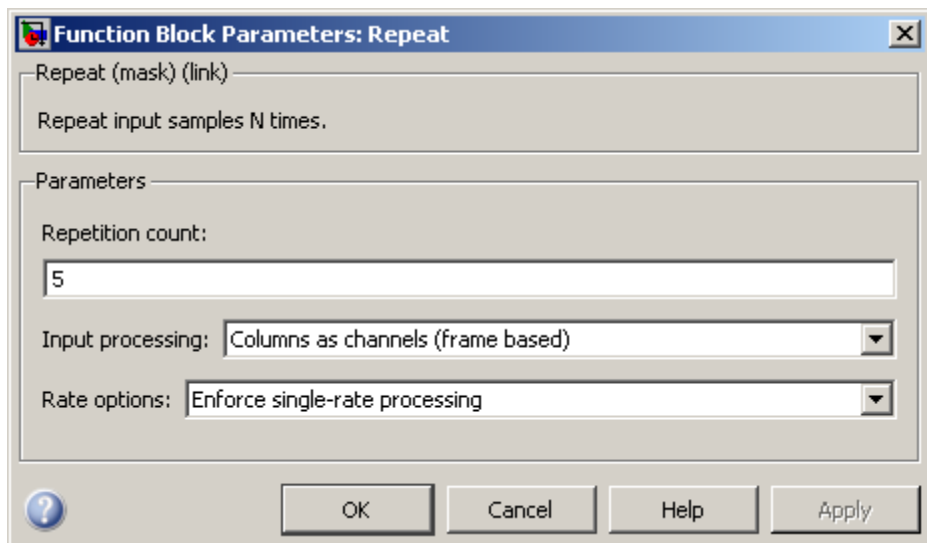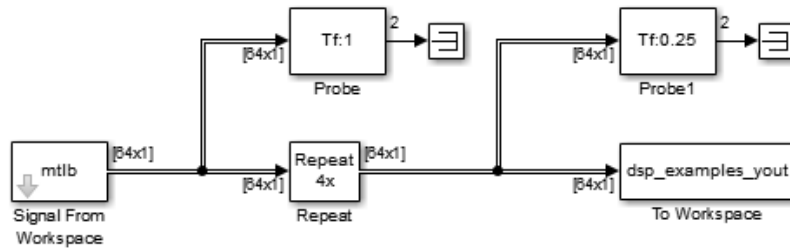
Repetition count

The integer number of times, *L*, that the input value is repeated at the output. This is the factor by which the block increases the output frame size or sample rate.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate

channel. In this mode, the block can perform single-rate or multirate processing.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. In this mode, the block always performs multirate processing.

Note The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

Specify the method by which the block upsamples the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate by increasing the output frame size by a factor of *L*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.

- `Allow multirate processing` — When you select this option, the block resamples the signal such that the output sample rate is *L* times faster than the input sample rate.

Initial conditions

The value with which the block is initialized for cases of nonzero latency; a scalar or matrix. This parameter appears only when you configure the block to perform multirate processing.

Repeat

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

FIR Interpolation	DSP System Toolbox
Upsample	DSP System Toolbox

Purpose   Compute filter estimates for input using RLS adaptive filter algorithm

Library   dspobslib

Description
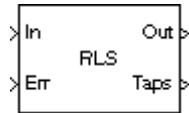
Note The RLS Adaptive Filter block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the RLS Filter block.

The RLS Adaptive Filter block recursively computes the recursive least squares (RLS) estimate of the FIR filter coefficients.

The corresponding RLS filter is expressed in matrix form as

$$
k(n) = \frac{\lambda^{-1} P(n-1) u(n)}{1 + \lambda^{-1} u^H(n) P(n-1) u(n)}
$$
$$
y(n) = \hat{w}^H(n-1) u(n)
$$
$$
e(n) = d(n) - y(n)
$$
$$
\hat{w}(n) = \hat{w}(n-1) + k(n) e^*(n)
$$
$$
P(n) = \lambda^{-1} P(n-1) - \lambda^{-1} k(n) u^H(n) P(n-1)
$$

where λ^{-1} denotes the reciprocal of the exponential weighting factor. The variables are as follows

Variable	Description
n	The current algorithm iteration
$u(n)$	The buffered input samples at step n
$P(n)$	The inverse correlation matrix at step n
$k(n)$	The gain vector at step n
$\hat{w}(n)$	The vector of filter-tap estimates at step n

Variable	Description
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
λ	The exponential memory weighting factor

The block icon has port labels corresponding to the inputs and outputs of the RLS algorithm. Note that inputs to the In and Err ports must be sample-based scalars. The signal at the Out port is a scalar, while the signal at the Taps port is a sample-based vector.

Block Ports	Corresponding Variables
In	u, the scalar input, which is internally buffered into the vector $u(n)$
Out	$y(n)$, the filtered scalar output
Err	$e(n)$, the scalar estimation error
Taps	$\hat{w}(0)$, the vector of filter-tap estimates

An optional Adapt input port is added when you select the **Adapt input** check box in the dialog box. When this port is enabled, the block continuously adapts the filter coefficients while the Adapt input is nonzero. A zero-valued input to the Adapt port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero Adapt input.

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the inverse correlation matrix $P(n)$. This decreases the total number of computations by a factor of two.

The **FIR filter length** parameter specifies the length of the filter that the RLS algorithm estimates. The **Memory weighting factor** corresponds to λ in the equations, and specifies how quickly the filter

"forgets" past sample information. Setting λ=1 specifies an infinite memory; typically, `0.95`≤λ≤`1`.

The **Initial value of filter taps** specifies the initial value $\overset{\frown}{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The initial value of $P(n)$ is

$$ I \frac{1}{\overset{\frown}{\sigma}^{2}} $$

where you specify $\overset{\frown}{\sigma}^{2}$ in the **Initial input variance estimate** parameter.

Examples   The `rlsdemo` example illustrates a noise cancellation system built around the RLS Adaptive Filter block.

RLS Adaptive Filter (Obsolete)

Dialog Box

Block Parameters: RLS Adaptive Filter

―RLS Adaptive Filter (mask)―

Exponentially weighted recursive least-squares (RLS) algorithm for adaptive FIR filtering of input signal. If Adapt input checkbox is enabled, and the Adapt input port is zero, the algorithm stops adapting the filter coefficients.

―Parameters―

FIR filter length:

32

Memory weighting factor (0 to 1):

1.0

Initial value of filter taps:

0.0

Initial input variance estimate:

0.1

☐ Adapt input

| OK | Cancel | Help | Apply |

FIR filter length
> The length of the FIR filter.

Memory weighting factor
> The exponential weighting factor, in the range [0,1]. A value of 1 specifies an infinite memory. Tunable.

Initial value of filter taps
> The initial FIR filter coefficients.

Initial input variance estimate
> The initial value of $1/P(n)$.

Adapt input
> Enables the Adapt port.

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

**Supported
Data
Types**
- Double-precision floating point
- Single-precision floating point

See Also

Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
LMS Adaptive Filter (Obsolete)	DSP System Toolbox

See "Adaptive Filters in Simulink" for related information.

RLS Filter

Purpose	Compute filtered output, filter error, and filter weights for given input and desired signal using RLS adaptive filter algorithm
Library	Filtering / Adaptive Filters dspadpt3

Description
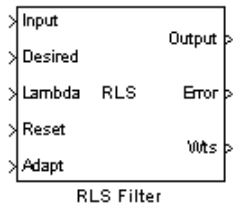
The RLS Filter block recursively computes the least squares estimate (RLS) of the FIR filter weights. The block estimates the filter weights, or coefficients, needed to convert the input signal into the desired signal. Connect the signal you want to filter to the Input port. The input signal can be a scalar or a column vector. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal. The Error port outputs the result of subtracting the output signal from the desired signal.

The corresponding RLS filter is expressed in matrix form as

$$\mathbf{k}(n) = \frac{\lambda^{-1}\mathbf{P}(n-1)\mathbf{u}(n)}{1 + \lambda^{-1}\mathbf{u}^{H}(n)\mathbf{P}(n-1)\mathbf{u}(n)}$$

$$y(n) = \mathbf{w}(n-1)\mathbf{u}(n)$$

$$e(n) = d(n) - y(n)$$

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mathbf{k}^{H}(n)e(n)$$

$$\mathbf{P}(n) = \lambda^{-1}\mathbf{P}(n-1) - \lambda^{-1}\mathbf{k}(n)\mathbf{u}^{H}(n)\mathbf{P}(n-1)$$

where λ^{-1} denotes the reciprocal of the exponential weighting factor. The variables are as follows

Variable	Description
n	The current time index
$\mathbf{u}(n)$	The vector of buffered input samples at step n
$\mathbf{P}(n)$	The inverse correlation matrix at step n

Variable	Description
$\mathbf{k}(n)$	The gain vector at step n
$\mathbf{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
λ	The forgetting factor

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the inverse correlation matrix $P(n)$. This decreases the total number of computations by a factor of two.

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

The **Forgetting factor (0 to 1)** parameter corresponds to λ in the equations. It specifies how quickly the filter "forgets" past sample information. Setting $\lambda=1$ specifies an infinite memory. Typically,

$1 - \dfrac{1}{2L} < \lambda < 1$, where L is the filter length. You can specify a forgetting factor using the input port, Lambda, or enter a value in the **Forgetting factor (0 to 1)** parameter in the Block Parameters: RLS Filter dialog box.

Enter the initial filter weights, $w(0)$, as a vector or a scalar for the **Initial value of filter weights** parameter. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

The initial value of $P(n)$ is

$$\frac{1}{\sigma^2} I$$

where you specify σ^2 in the **Initial input variance estimate** parameter.

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

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

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

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

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

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

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

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

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

Falling edge   Falling edge

Falling edge   Falling edge   **Not a falling edge** because it continues a fall from a positive value to zero.

- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge`, as described above

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

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

Examples   The `rlsdemo` example illustrates a noise cancellation system built around the RLS Filter block.

RLS Filter

Block Parameters: RLS Filter

RLS Adaptive filter (mask) (link)

Computes filter weights based on the exponentially weighted recursive least-squares (RLS) algorithm for adaptive filtering of the input signal.

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

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

Parameters

Filter length:

32

Specify forgetting factor via: Dialog

Forgetting factor (0 to 1):

1.0

Initial value of filter weights:

0

Initial input variance estimate:

0.1

☐ Adapt port

Reset port: None

☑ Output filter weights

OK   Cancel   Help   Apply

Dialog Box

Filter length

Enter the length of the FIR filter weights vector.

Specify forgetting factor via

Select Dialog to enter a value for the forgetting factor in the Block parameters: RLS Filter dialog box. Select Input port to specify the forgetting factor using the Lambda input port.

Forgetting factor (0 to 1)

Enter the exponential weighting factor in the range $0 \leq \lambda \leq 1$. A value of 1 specifies an infinite memory. Tunable.

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Initial input variance estimate

The initial value of $1/P(n)$.

Adapt port

Select this check box to enable the Adapt input port.

Reset input

Select this check box to enable the Reset input port.

Output filter weights

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

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

Supported Data Types

• Double-precision floating point

• Single-precision floating point

See Also

Kalman Adaptive Filter (Obsolete)	DSP System Toolbox
LMS Filter	DSP System Toolbox
Block LMS Filter	DSP System Toolbox
Fast Block LMS Filter	DSP System Toolbox

See "Adaptive Filters in Simulink" for related information.

RMS

Purpose   Compute root-mean-square value of input or sequence of inputs

Library   Statistics

        dspstat3

Description

The RMS block computes the RMS value of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input. The RMS block can also track the RMS value in a sequence of inputs over a period of time. The **Running RMS** parameter selects between basic operation and running operation.

Basic Operation

When you do not select the **Running RMS** check box, the block computes the RMS value of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input at each individual sample time, and outputs the array *y*. Each element in *y* is the RMS value of the corresponding column, row, vector, or entire input. The output *y* depends on the setting of the **Find the RMS value over** parameter. For example, consider a 3-dimensional input signal of size *M*-by-*N*-by-*P*:

- `Entire input` — The output at each sample time is a scalar that contains the RMS value of the entire input.

- `Each row` — The output at each sample time consists of an *M*-by-1-by-*P* array, where each element contains the RMS value of each vector over the second dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is an *M*-by-1 column vector.

- `Each column` — The output at each sample time consists of a 1-by-*N*-by-*P* array, where each element contains the RMS value of each vector over the first dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is a 1-by-*N* row vector.

In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- Specified dimension — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as that when you select Each column. If **Dimension** is set to 2, the output is the same as when you select Each row. If **Dimension** is set to 3, the output at each sample time is an *M*-by-*N* matrix containing the RMS value of each vector over the third dimension of the input.

The RMS value of the *j*th column of an *M*-by-*N* input matrix *u* is given by

$$y_j = \sqrt{\frac{\sum_{i=1}^{M} |u_{ij}|^2}{M}} \qquad 1 \le j \le N$$

```
y = sqrt(sum(u.*conj(u))/size(u,1))    % Equivalent MATLAB code
```

Running Operation

When you select the **Running RMS** check box, the block tracks the RMS value of successive inputs to the block. In this mode, you must also specify a value for the **Input processing** parameter:

- When you select Elements as channels (sample based), the block outputs an *M*-by-*N* array. Each element y_{ij} of the output contains the RMS value of the element u_{ij} over all inputs since the last reset.

- When you select Columns as channels (frame based), the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains 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.

Running Operation for Variable-Size Inputs

When your inputs are of variable size, and you select the **Running RMS** check box, there are two options:

- If you set the **Input processing** parameter to `Elements as channels (sample based)`, the state is reset.

- If you set the **Input processing** parameter to `Columns as channels (frame based)`, then there are two cases:

  - When the input size difference is in the number of channels (i.e., number of columns), the state is reset.

  - When the input size difference is in the length of channels (i.e., number of rows), there is no reset and the running operation is carried out as usual.

Resetting the Running RMS

The block resets the running RMS whenever a reset event is detected at the optional `Rst` port. The reset sample time must be a positive integer multiple of the input sample time.

When a reset event occurs while the **Input processing** parameter is set to `Elements as channels (sample based)`, the running RMS for each channel is initialized to the value in the corresponding channel of the current input. Similarly, when the **Input processing** parameter is set to `Columns as channels (frame based)`, the running RMS for each channel is initialized to the earliest value in each channel of the current input.

You specify the reset event in the **Reset port** parameter:

- `None` disables the `Rst` port.

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

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

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

Not a rising edge because it continues a rise from a negative value to zero.

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

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

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

Not a falling edge because it continues a fall from a positive value to zero.

- Either edge — Triggers a reset operation when the Rst input is a Rising edge or Falling edge, as described earlier

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

RMS

> **Note** When running simulations in the Simulink MultiTasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

Examples   In the following ex_rms_ref model, the RMS block calculates the running RMS of a 3-by-2 matrix input, u. The **Input processing** parameter is set to Columns as channels (frame based), so the block processes the input as a two channel signal with a frame size of three. The running RMS is reset at *t*=2 by an impulse to the block's Rst port.

The operation of the block is shown in the following figure.

RMS
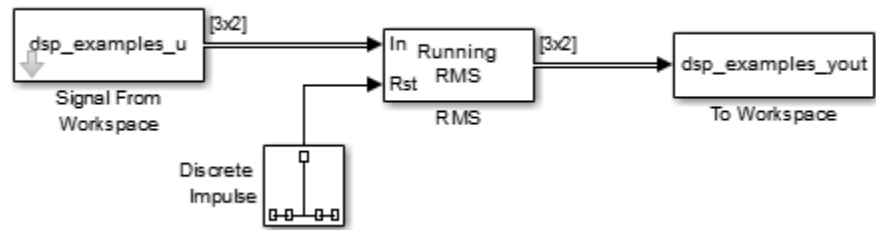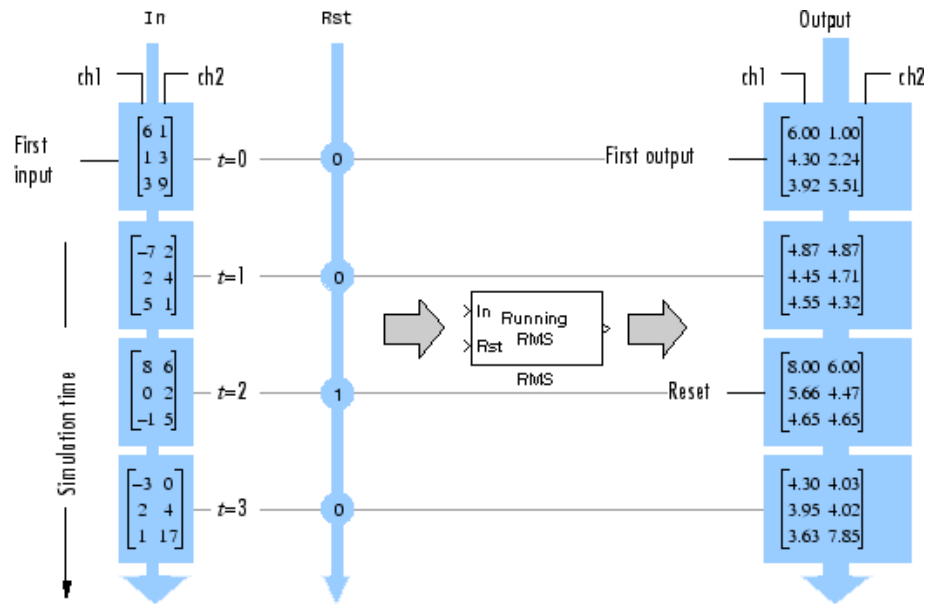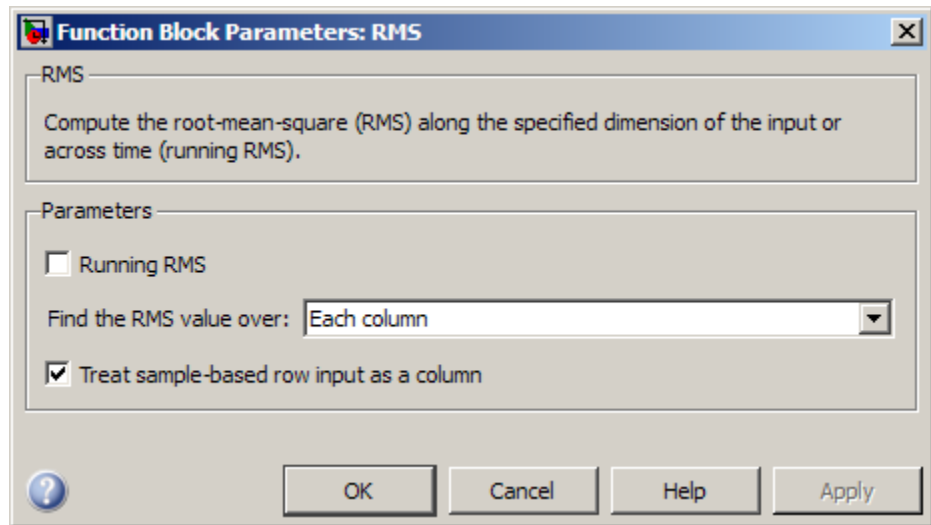
Running RMS

Enables running operation when selected.

Input processing

Specify how the block should process the input when computing the running RMS. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

This parameter appears only when you select the **Running RMS** check box.

> **Note** The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Reset port
> Specify the reset event that causes the block to reset the running RMS. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you select the **Running RMS** check box. For more information, see "Resetting the Running RMS" on page 2-1292.

Find the RMS value over
> Specify whether to find the RMS value along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see "Basic Operation" on page 2-1290.

Treat sample-based row input as a column
> Select to treat sample-based length-M row vector inputs as M-by-1 column vectors. This parameter is only visible when the **Find the RMS value over** parameter is set to `Each column`.

> **Note** This check box will be removed in a future release. See "Sample-Based Row Vector Processing Changes" in the DSP System Toolbox Release Notes.

Dimension
> Specify the dimension (one-based value) of the input signal, over which the RMS value is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the RMS value over** parameter is set to `Specified dimension`.

RMS

Supported Data Types	• Double-precision floating point
	• Single-precision floating point
	• Boolean — The block accepts Boolean inputs to the Rst port.

| **See Also** | Mean | DSP System Toolbox |
| | Variance | DSP System Toolbox |

Purpose   Sample and hold input signal

Library   Signal Operations

dspsigops
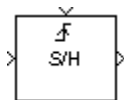
Description   The Sample and Hold block acquires the input at the signal port whenever it receives a trigger event at the trigger port (marked by ⌁). The block then holds the output at the acquired input value until the next triggering event occurs.

The trigger input must be a sample-based scalar with sample rate equal to the input frame rate at the signal port. You specify the trigger event using the **Trigger type** parameter:

- Rising edge triggers the block to acquire the signal input when the trigger input rises from a negative value or zero to a positive value.

- Falling edge triggers the block to acquire the signal input when the trigger input falls from a positive value or zero to a negative value.

- Either edge triggers the block to acquire the signal input when the trigger input either rises from a negative value or zero to a positive value or falls from a positive value or zero to a negative value.

You specify the block's output prior to the first trigger event using the **Initial condition** parameter. When the acquired input is an M-by-N matrix, the **Initial condition** can be an M-by-N matrix, or a scalar to be repeated across all elements of the matrix. When the input is a length-M unoriented vector, the **Initial condition** can be a length-M row or column vector, or a scalar to be repeated across all elements of the vector.

If you select the **Latch (buffer) input** check box, the block outputs the value of the input from the previous time step until the next triggering event occurs. To use this block in a loop, select this check box.

Sample and Hold
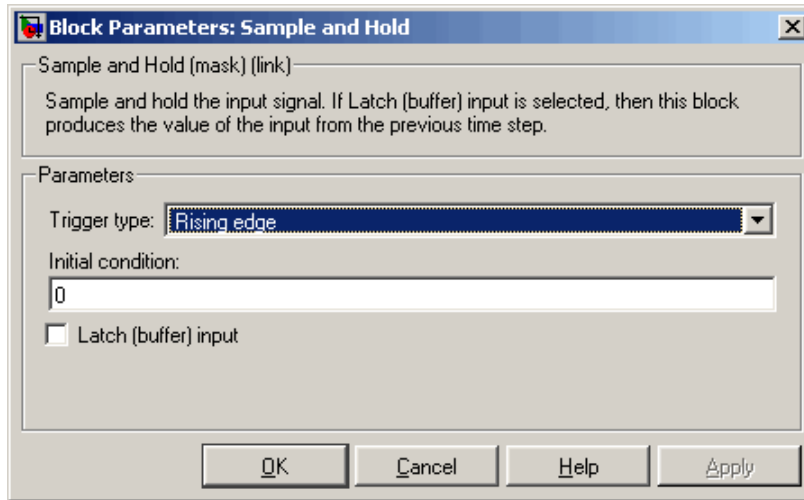
Dialog Box

Trigger type
> The type of event that triggers the block to acquire the input signal.

Initial condition
> The block's output prior to the first trigger event.

Latch (buffer) input
> If you select this check box, the block outputs the value of the input from the previous time step until the next triggering event occurs.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Trigger	• Any data type supported by the Trigger block
Outputs	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Downsample	DSP System Toolbox
N-Sample Switch	DSP System Toolbox

Scalar Quantizer (Obsolete)

Purpose   Convert input signal into set of quantized output values or index values, or convert set of index values into quantized output signal

Library   dspobslib

Description
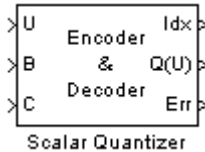
Scalar Quantizer

Note The Scalar Quantizer block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Scalar Quantizer Encoder block or the Scalar Quantizer Decoder block.

The Scalar Quantizer block has three modes of operation. In `Encoder` mode, the block maps each input value to a quantization region by comparing the input value to the quantizer boundary points defined in the **Boundary points** parameter. The block outputs the index of the associated region. In `Decoder` mode, the block transforms the input index values into quantized output values, defined in the **Codebook** parameter. In the `Encoder and Decoder` mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

You can select how you want to enter the **Boundary points** and/or **Codebook** values using the **Source of quantizer** parameters. When you select `Specify via dialog`, type the parameters into the block parameters dialog box. Select `Input ports`, and port B and/or C appears on the block. In `Encoder` and `Encoder and decoder` mode, the input to port B is used as the **Boundary points**. In `Decoder` and `Encoder and decoder` mode, the input to port C is used as the **Codebook**.

In `Encoder` and `Encoder and decoder` mode, the **Boundary points** are the values used to break up the input signal into regions. Each region is specified by an index number. When your first boundary point is `-inf` and your last boundary point is `inf`, your quantizer is unbounded. When your first and last boundary point is finite, your

quantizer is bounded. When only your first or last boundary point is
-inf or inf, your quantizer is semi-bounded.

For instance, when your input signal ranges from 0 to 11, you can create
a bounded quantizer using the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The boundary points can have equal or varied spacing. Any input
values between 0 and 0.5 would correspond to index 0. Input values
between 0.5 and 3.7 would correspond to index 1, and so on.

Suppose you wanted to create an unbounded quantizer with the
following boundary points:

```
[-inf 0 2 5.5 7.1 10 inf]
```

When your input signal has values less than 0, these values would be
assigned to index 0. When your input signal has values greater than
10, these values would be assigned to index 6.

When an input value is the same as a boundary point, the **Tie-breaking
rule** parameter defines the index to which the value is assigned. When
you want the input value to be assigned to the lower index value, select
Choose the lower index. To assign the input value with the higher
index, select Choose the higher index.

In Decoder and Encoder and decoder mode, the **Codebook** is a vector
of quantized output values that correspond to each index value.

In Encoder and Encoder and decoder mode, the **Searching method**
determines how the appropriate quantizer index is found. Select Linear
and the Scalar Quantizer block compares the input value to the first
region defined by the first two boundary points. When the input value
does not fall within this region, the block then compares the input
value to the next region. This process continues until the input value is
determined to be within a region and is associated with the appropriate
index value. The computational cost of this process is of the order P,
where P is the number of boundary points.

Scalar Quantizer (Obsolete)

Select `Binary` for the **Searching method** and the block compares the input value to the middle value of the boundary points vector. When the input value is larger than this boundary point, the block discards the boundary points that are lower than this middle value. The block then compares the input value to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the input value is associated with the appropriate index value. The computational cost of this process is of the order $\log_2 P$, where P is the number of boundary points. In most cases, the `Binary` option is faster than the `Linear` option.

In `Decoder` mode, the input to this block is a vector of index values, where $0 \leq \text{index} < N$ and N is the length of the codebook vector. Use the **Action for out of range input** parameter to determine what happens when an input index value is out of this range. When you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N-1, select `Clip`. When you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to N are set to N-1, select `Clip and warn`. When you want the simulation to stop and display an error when the index values are out of range, select `Error`.

In `Encoder and decoder` mode, you can select the **Output the quantization error** check box. The quantization error is the difference between the input value and the quantized output value. Select this check box to output the quantization error for each input value from the Err port on this block.

Data Type Support

In `Encoder` mode, the input data values and the boundary points can be the input to the block at ports U and B. Similarly, in `Encoder and decoder` mode, the codebook values can also be the input to the block at port C. The data type of the input data values, boundary points, and codebook values can be `double`, `single`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. In `Decoder` mode, the input to the block can be the index values and the codebook values. The data type of the index input to the block at port Idx can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or

`int32`. The data type of the codebook values can be `double`, `single`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`.

In `Encoder` mode, the output of the block is the index values. In `Encoder and decoder` mode, the output can also include the quantized output values and the quantization error. In `Encoder` and `Encoder and decoder` mode, use the **Output index data type** parameter to specify the data type of the index output from the block at port Idx. The data type of the index output can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. The data type of the quantized output and the quantization error can be `double`, `single`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. In `Decoder` mode, the output of the block is the quantized output values. Use the **Output data type** parameter to specify the data type of the quantized output values. The data type can be `double`, `single`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`.

Note The input data, codebook values, boundary points, quantization error, and the quantized output values must have the same data type whenever they are present in any of the quantizer modes.

Scalar Quantizer (Obsolete)

Dialog Box

Block Parameters: Scalar Quantizer

Scalar Quantizer (mask) (link)

The Scalar Quantizer block has three modes of operation. In Encoder mode, the block maps each input value to a quantization region by comparing the input value to the user-specified quantizer boundary points. The block outputs the index of the associated region. In Decoder mode, the block transforms the input index values into quantized output values, defined by the quantizer codebook. In the Encoder & Decoder mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

The Boundary points parameter is a vector of dimension N whose values are arranged in ascending order ([p1 p2 ... pN]). The quantizer codebook is a vector of length (N-1). The quantizer is unbounded if p1 is -inf and pN is inf. When the quantizer is bounded, p1 is the lower bound for the input values. Any input less than p1 is clipped to p1 and then quantized. Similarly, pN is the upper bound for the input values. Any input greater than pN is clipped to pN and is then quantized.

Parameters

Quantizer mode: Encoder

Source of quantizer parameters: Specify via dialog

Boundary points:

[0 1 2 3 4 5 6 7 8 9]

Searching method: Binary

Tie-breaking rule: Choose the lower index

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

Output index data type: int32

OK   Cancel   Help   Apply

Block Parameters: Scalar Quantizer

Scalar Quantizer (mask) (link)

The Scalar Quantizer block has three modes of operation.
In Encoder mode, the block maps each input value to a quantization region by comparing the input value to the user-specified quantizer boundary points. The block outputs the index of the associated region. In Decoder mode, the block transforms the input index values into quantized output values, defined by the quantizer codebook. In the Encoder & Decoder mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

The Boundary points parameter is a vector of dimension N whose values are arranged in ascending order ([p1 p2 ... pN]). The quantizer codebook is a vector of length (N-1). The quantizer is unbounded if p1 is -inf and pN is inf. When the quantizer is bounded, p1 is the lower bound for the input values. Any input less than p1 is clipped to p1 and then quantized. Similarly, pN is the upper bound for the input values. Any input greater than pN is clipped to pN and is then quantized.

Parameters

Quantizer mode: | Decoder |

Source of quantizer parameters: | Specify via dialog |

Codebook:

| [0.0   1.5   2.5   3.5   4.5   5.5   6.5 7.5 8.5] |

Action for out of range input: | Clip and warn |

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

Output data type: | double |

| OK | Cancel | Help | Apply |

Scalar Quantizer (Obsolete)

Block Parameters: Scalar Quantizer

Scalar Quantizer (mask) (link)

The Scalar Quantizer block has three modes of operation.
In Encoder mode, the block maps each input value to a quantization region by comparing the input value to the user-specified quantizer boundary points. The block outputs the index of the associated region.
In Decoder mode, the block transforms the input index values into quantized output values, defined by the quantizer codebook.
In the Encoder & Decoder mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

The Boundary points parameter is a vector of dimension N whose values are arranged in ascending order ([p1 p2 ... pN]). The quantizer codebook is a vector of length (N-1). The quantizer is unbounded if p1 is -inf and pN is inf. When the quantizer is bounded, p1 is the lower bound for the input values. Any input less than p1 is clipped to p1 and then quantized. Similarly, pN is the upper bound for the input values. Any input greater than pN is clipped to pN and is then quantized.

Parameters

Quantizer mode: Encoder and decoder

Source of quantizer parameters: Specify via dialog

Boundary points:
[0 1 2 3 4 5 6 7 8 9]

Codebook:
[0.0 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5]

Searching method: Binary

Tie-breaking rule: Choose the lower index

☑ Output the quantization error

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

Output index data type: int32

OK   Cancel   Help   Apply

Quantizer mode

Specify `Encoder`, `Decoder`, or `Encoder and decoder` as a mode of operation.

Source of quantizer parameters

Choose `Specify via dialog` to type the parameters into the block parameters dialog box. Select `Input ports` to specify the parameters using the block's input ports. In `Encoder` and `Encoder and decoder` mode, input the **Boundary points** using port B. In `Decoder` and `Encoder and decoder` mode, input the **Codebook** values using port C.

Boundary points

Enter a vector of values that represent the boundary points of the quantizer regions. Tunable.

Codebook

Enter a vector of quantized output values that correspond to each index value. Tunable.

Searching method

Select `Linear` and the block finds the region in which the input value is located using a linear search. Select `Binary` and the block finds the region in which the input value is located using a binary search.

Tie-breaking rule

Set this parameter to determine the behavior of the block when the input value is the same as the boundary point. When you select `Choose the lower index`, the input value is assigned to lower index value. When you select `Choose the higher index`, the value is assigned to the higher index.

Action for out of range input

Choose the block's behavior when an input index value is out of range, where $0 \leq$ index $< N$ and N is the length of the codebook vector. Select `Clip`, when you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N-1. Select `Clip and warn`, when you want to be warned when any index values less than 0 are set to 0 and any

Scalar Quantizer (Obsolete)

index values greater than or equal to *N* are set to *N* -1. Select Error, when you want the simulation to stop and display an error when the index values are out of range.

Output the quantization error

In Encoder and decoder mode, select this check box to output the quantization error from the Err port on this block.

Output index data type

In Encoder and Encoder and decoder mode, specify the data type of the index output from the block at port Idx. The data type can be uint8, uint16, uint32, int8, int16, or int32. This parameter becomes visible when you select the **Show additional parameters** check box.

Output data type

In Decoder mode, specify the data type of the quantized output. The data type can be uint8, uint16, uint32, int8, int16, int32, single, or double. This parameter becomes visible when you select Specify via dialog for the **Source of quantizer parameters** and you select the **Show additional parameters** check box.

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types
- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

For more information on what data types are supported for each quantizer mode, see "Data Type Support" on page 2-1304.

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Design	DSP System Toolbox
Scalar Quantizer Encoder	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox

Scalar Quantizer Decoder

Purpose   Convert each index value into quantized output value

Library   Quantizers

dspquant2
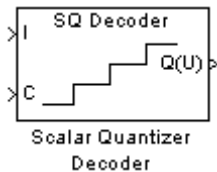
Description   The Scalar Quantizer Decoder block transforms the zero-based input index values into quantized output values. The set of all possible quantized output values is defined by the **Codebook values** parameter.

Scalar Quantizer
Decoder

Use the **Codebook values** parameter to specify a matrix containing all possible quantized output values. You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select Specify via dialog, type the codebook values into the block parameters dialog box. When you select Input port, port C appears on the block. The block uses the input to port C as the **Codebook values** parameter.

The input to this block is a vector of integer index values, where 0 ≤ index < N and N is the number of distinct codeword vectors in the codebook matrix. Use the **Action for out of range index value** parameter to determine what happens when an input index value is outside this range. When you want any index value less than 0 to be set to 0 and any index value greater than or equal to N to be set to N-1, select Clip. When you want to be warned when clipping occurs, select Clip and warn. When you want the simulation to stop and the block to display an error when the index values are out of range, select Error.

Data Type Support

The data type of the index values input at port I can be uint8, uint16, uint32, int8, int16, or int32. The data type of the codebook values input at port C can be double, single, or Fixed-point.

The output of the block is the quantized output values. If, for the **Source of codebook** parameter, you select Specify via dialog, the **Codebook and output data type** parameter appears. You can use this parameter to specify the data type of the codebook and quantized output values. In this case, the data type of the output values can be Same as input, double, single, Fixed-point, or User-defined.

If, for the **Source of codebook** parameter you select Input port, the quantized output values have the same data type as the codebook values input at port C.

Dialog Box

The **Main** pane of the Scalar Quantizer Decoder block dialog appears as follows.
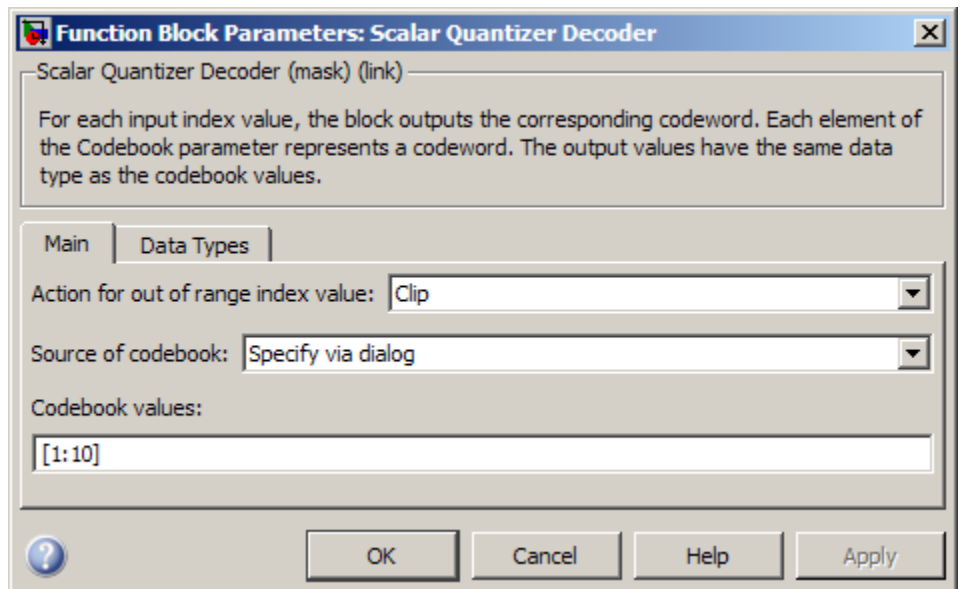
Action for out of range index value

Use this parameter to determine the block's behavior when an input index value is out of range, where $0 \leq$ index $<N$ and N is the length of the codebook vector. Select Clip, when you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N -1. Select Clip and warn, when you want to be warned when clipping occurs. Select Error, when you want the simulation to stop and the block to display an error when the index values are outside the range.

Scalar Quantizer Decoder

Source of codebook

Choose `Specify via dialog` to type the codebook values into the block parameters dialog box. Select `Input port` to specify the codebook using input port C.

Codebook values

Enter a vector of quantized output values that correspond to each index value. Tunable.

The **Data Types** pane of the Scalar Quantizer Decoder block dialog appears as follows.

Codebook and output data type

Specify the data type of the codebook and quantized output values. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit:  Same as input`.

- A built in data type, such as double

- An expression that evaluates to a valid data type, for example, fixdt(1,16)

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

This parameter is available only when you set the **Source of codebook** parameter to Specify via dialog. If you set the **Source of codebook** parameter to Input port, the output values have the same data type as the input codebook values.

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

Port	Supported Data Types
I	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
C	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
Q(U)	• Double-precision floating point
	• Single-precision floating point
	• Fixed point

Scalar Quantizer Decoder

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

For more information on what data types are supported for each quantizer mode, see "Data Type Support" on page 2-1312.

See Also

Quantizer	Simulink
Scalar Quantizer Design	DSP System Toolbox
Scalar Quantizer Encoder	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox

Purpose   Start Scalar Quantizer Design Tool (SQDTool) to design scalar quantizer using Lloyd algorithm

Library   Quantizers

dspquant2
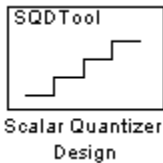
Description   Double-click on the Scalar Quantizer Design block to start SQDTool, a GUI that allows you to design and implement a scalar quantizer. Based on your input values, SQDTool iteratively calculates the codebook values that minimize the mean squared error until the stopping criteria for the design process is satisfied. The block uses the resulting quantizer codebook values and boundary points to implement your scalar quantizer encoder and/or decoder.

For the **Training Set** parameter, enter a set of observations, or samples, of the signal you want to quantize. This data can be any variable defined in the MATLAB workspace including a variable created using a MATLAB function, such as the default value randn(10000,1).

You have two choices for the **Source of initial codebook** parameter. Select Auto-generate to have the block choose the values of the initial codebook vector. In this case, the minimum training set value becomes the first codeword, and the maximum training set value becomes the last codeword. Then, the remaining initial codewords are equally spaced between these two values to form a codebook vector of length N, where N is the **Number of levels** parameter. When you select User defined, enter the initial codebook values in the **Initial codebook** field. Then, set the **Source of initial boundary points** parameter. You can select Mid-points to locate the boundary points at the midpoint between the codewords. To calculate the mid-points, the block internally arranges the initial codebook values in ascending order. You can also choose User defined and enter your own boundary points in the **Initial boundary points (unbounded)** field. Only one boundary point can be located between two codewords. When you select User defined for the **Source of initial boundary points** parameter, the values you enter in the **Initial codebook** and **Initial boundary points (unbounded)** fields must be arranged in ascending order.

Scalar Quantizer Design

Note This block assumes that you are designing an unbounded quantizer. Therefore, the first and last boundary points are always `-inf` and `inf` regardless of any other boundary point values you might enter.

After you have specified the quantization parameters, the block performs an iterative process to design the optimal scalar quantizer. Each step of the design process involves using the Lloyd algorithm to calculate codebook values and quantizer boundary points. Then, the block calculates the squared quantization error and checks whether the stopping criteria has been satisfied.

The two possible options for the **Stopping criteria** parameter are `Relative threshold` and `Maximum iteration`. When you want the design process to stop when the fractional drop in the squared quantization error is below a certain value, select `Relative threshold`. Then, for **Relative threshold**, type the maximum acceptable fractional drop. When you want the design process to stop after a certain number of iterations, choose `Maximum iteration`. Then, enter the maximum number of iterations you want the block to perform in the **Maximum iteration** field. For **Stopping criteria**, you can also choose `Whichever comes first` and enter a **Relative threshold** and **Maximum iteration** value. The block stops iterating as soon as one of these conditions is satisfied.

With each iteration, the block quantizes the training set values based on the newly calculated codebook values and boundary points. When the training point lies on a boundary point, the algorithm uses the **Tie-breaking rules** parameter to determine which region the value is associated with. When you want the training point to be assigned to the lower indexed region, select `Lower indexed codeword`. To assign the training point with the higher indexed region, select `Higher indexed codeword`.

The **Searching methods** parameter determines how the block compares the training points to the boundary points. Select `Linear search` and SQDTool compares each training point to each quantization

region sequentially. This process continues until all the training points are associated with the appropriate regions.

Select `Binary search` for the **Searching methods** parameter and the block compares the training point to the middle value of the boundary points vector. When the training point is larger than this boundary point, the block discards the lower boundary points. The block then compares the training point to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the training point is associated with the appropriate region.

Click **Design and Plot** to design the quantizer with the parameter values specified on the left side of the GUI. The performance curve and the staircase character of the quantizer are updated and displayed in the figures on the right side of the GUI.

Note You must click **Design and Plot** to apply any changes you make to the parameter values in the SQDTool dialog box.

SQDTool can export parameter values that correspond to the figures displayed in the GUI. Click the **Export Outputs** button, or press **Ctrl+E**, to export the **Final Codebook**, **Final Boundary Points**, and **Error** values to the workspace, a text file, or a MAT-file. The **Error** values represent the mean squared error for each iteration.

In the **Model** section of the GUI, specify the destination of the block that will contain the parameters of your quantizer. For **Destination**, select `Current model` to create a block with your parameters in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Select `New model` to create a block in a new model file.

From the **Block type** list, select `Encoder` to design a Scalar Quantizer Encoder block. Select `Decoder` to design a Scalar Quantizer Decoder block. Select `Both` to design a Scalar Quantizer Encoder block and a Scalar Quantizer Decoder block.

Scalar Quantizer Design

In the **Encoder block name** field, enter a name for the Scalar Quantizer Encoder block. In the **Decoder block name** field, enter a name for the Scalar Quantizer Decoder block. When you have a Scalar Quantizer Encoder and/or Decoder block in your destination model with the same name, select the **Overwrite target block(s)** check box to replace the block's parameters with the current parameters. When you do not select this check box, a new Scalar Quantizer Encoder and/or Decoder block is created in your destination model.

Click **Generate Model**. SQDTool uses the parameters that correspond to the current plots to set the parameters of the Scalar Quantizer Encoder and/or Decoder blocks.

Dialog Box

Scalar Quantizer Design

Training Set

Enter the samples of the signal you would like to quantize. This data set can be a MATLAB function or a variable defined in the MATLAB workspace. The typical length of this data vector is `1e6`.

Source of initial codebook

Select `Auto-generate` to have the block choose the initial codebook values. Select `User defined` to enter your own initial codebook values.

Number of levels

Enter the length of the codebook vector. For a b-bit quantizer, the length should be $N = 2^b$.

Initial codebook

Enter your initial codebook values. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter.

Source of initial boundary points

Select `Mid-points` to locate the boundary points at the midpoint between the codebook values. Choose `User defined` to enter your own boundary points. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter.

Initial boundary points (unbounded)

Enter your initial boundary points. This block assumes that you are designing an unbounded quantizer. Therefore, the first and last boundary point are `-inf` and `inf`, regardless of any other boundary point values you might enter. From the **Source of initial boundary points** list, select `User defined` in order to activate this parameter.

Stopping criteria

Choose `Relative threshold` to enter the maximum acceptable fractional drop in the squared quantization error. Choose `Maximum iteration` to specify the number of iterations at which to stop. Choose `Whichever comes first` and the block stops the iteration process as soon as the relative threshold or maximum iteration value is attained.

Relative threshold

    Type the value that is the maximum acceptable fractional drop in the squared quantization error.

Maximum iteration

    Enter the maximum number of iterations you want the block to perform. From the **Stopping criteria** list, select `Maximum iteration` in order to activate this parameter.

Searching methods

    Choose `Linear search` to use a linear search method when comparing the training points to the boundary points. Choose `Binary search` to use a binary search method when comparing the training points to the boundary points.

Tie-breaking rules

    When a training point lies on a boundary point, choose `Lower indexed codeword` to assign the training point to the lower indexed quantization region. Choose `Higher indexed codeword` to assign the training point to the higher indexed region.

Design and Plot

    Click this button to display the performance curve and the staircase character of the quantizer in the figures on the right side of the GUI. These plots are based on the current parameter settings.

    You must click **Design and Plot** to apply any changes you make to the parameter values in the SQDTool GUI.

Export Outputs

    Click this button, or press **Ctrl+E**, to export the **Final Codebook**, **Final Boundary Points**, and **Error** values to the workspace, a text file, or a MAT-file.

Destination

    Choose `Current model` to create a Scalar Quantizer block in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Choose `New model` to create a block in a new model file.

Scalar Quantizer Design

Block type

Select `Encoder` to design a Scalar Quantizer Encoder block. Select `Decoder` to design a Scalar Quantizer Decoder block. Select `Both` to design a Scalar Quantizer Encoder block and a Scalar Quantizer Decoder block.

Encoder block name

Enter a name for the Scalar Quantizer Encoder block.

Decoder block name

Enter a name for the Scalar Quantizer Decoder block.

Overwrite target block(s)

When you do not select this check box and a Scalar Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, a new Scalar Quantizer Encoder and/or Decoder block is created in the destination model. When you select this check box and a Scalar Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, the parameters of these blocks are overwritten by new parameters.

Generate Model

Click this button and SQDTool uses the parameters that correspond to the current plots to set the parameters of the Scalar Quantizer Encoder and/or Decoder blocks.

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

- Double-precision floating point

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox

Scalar Quantizer Encoder   DSP System Toolbox

Uniform Encoder   DSP System Toolbox

Uniform Decoder   DSP System Toolbox

Scalar Quantizer Encoder

Purpose   Encode each input value by associating it with index value of quantization region

Library   Quantizers

dspquant2

Description

Scalar Quantizer Encoder
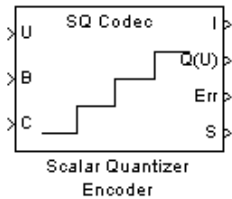
The Scalar Quantizer Encoder block maps each input value to a quantization region by comparing the input value to the quantizer boundary points defined in the **Boundary points** parameter. The block outputs the zero-based index of the associated region.

You can select how you want to enter the **Boundary points** using the **Source of quantizer parameters**. When you select Specify via dialog, type the boundary points into the block parameters dialog box. When you select Input port, port B appears on the block. The block uses the input to port B as the **Boundary points** parameter.

Use the **Boundary points** parameter to specify the boundary points for your quantizer. These values are used to break up the set of input numbers into regions. Each region is specified by an index number.

Let N be the number of quantization regions. When the codebook is defined as [c1 c2 c3 ... cN], and the **Boundary points** parameter is defined as [p0 p1 p2 p3 ... pN], then p0<c1<p1<c2 ... p(N-1)<cN<pN for a regular quantizer. When your quantizer is bounded, from the **Partitioning** list, select Bounded. You need to specify N+1 boundary points, or [p0 p1 p2 p3 ... pN]. When your quantizer is unbounded, from the **Partitioning** list, select Unbounded. You need to specify N-1 boundary points, or[p1 p2 p3 ... p(N-1)]; the block sets p0 equal to -inf and pN equal to inf.

The block uses the **Partitioning** parameter to interpret the boundary points you enter. For instance, to create a bounded quantizer, from the **Partitioning** list, select Bounded and enter the following boundary points:

[0 0.5 3.7 5.8 6.0 11]

The block assigns any input values between 0 and 0.5 to index 0, input values between 0.5 and 3.7 to index 1, and so on. The block assigns any values that are less than 0 to index 0, the lowest index value. The block assigns any values that are greater than 11 to index 4, the highest index value.

To create an unbounded quantizer, from the **Partitioning** list, select Unbounded and enter the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The block assigns any input values between 0 and 0.5 to index 1, input values between 0.5 and 3.7 to index 2, and so on. The block assigns any input values less than 0 to index 0 and any values greater than 11 to index 6.

The **Searching method** parameter determines how the appropriate quantizer index is found. When you select Linear, the Scalar Quantizer Encoder block compares the input value to the first region defined by the first two boundary points. When the input value does not fall within this region, the block then compares the input value to the next region. This process continues until the input value is determined to be within a region and is associated with the appropriate index value. The computational cost of this process is of the order P, where P is the number of boundary points.

When you select Binary for the **Searching method**, the block compares the input value to the middle value of the boundary points vector. When the input value is larger than this boundary point, the block discards the boundary points that are lower than this middle value. The block then compares the input value to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the input value is associated with the appropriate index value. The computational cost of this process is of the order $\log_2 P$, where P is the number of boundary points. In most cases, the Binary option is faster than the Linear option.

When an input value is the same as a boundary point, the **Tie-breaking rule** parameter determines the region to which the value is assigned.

Scalar Quantizer Encoder

When you want the input value to be assigned to the lower indexed region, select `Choose the lower index`. To assign the input value with the higher indexed region, select `Choose the higher index`.

Select the **Output codeword** check box to output the codeword values that correspond to each index value at port Q(U).

Select the **Output the quantization error** check box to output the quantization error for each input value from the Err port on this block. The quantization error is the difference between the input value and the quantized output value.

When you select either the **Output codeword** check box or the **Output quantization error** check box, you must also enter your codebook values. If, from the **Source of quantizer parameters** list, you choose `Specify via dialog`, use the **Codebook** parameter to enter a vector of quantized output values that correspond to each region. If, from the **Source of quantizer parameters** list, you choose `Input port`, use input port C to specify your codebook values.

If, for the **Partitioning** parameter, you select `Bounded`, the **Output clipping status** check box and the **Action for out of range input** parameter appear. When you select the **Output clipping status** check box, port S appears on the block. Any time an input value is outside the range defined by the **Boundary points** parameter, the block outputs a 1 at the S port. When the value is inside the range, the blocks outputs a 0.

You can use the **Action for out of range input** parameter to determine the block's behavior when an input value is outside the range defined by the **Boundary points** parameter. Suppose the boundary points for a bounded quantizer are defined as `[p0 p1 p2 p3 ...  pN]` and the possible index values are defined as `[i0 i1 i2 ...  i(N-1)]`, where `i0=0` and `i0<i1<i2<...<i(N-1)`. When you want any input value less than `p0` to be assigned to index value `i0` and any input values greater than `pN` to be assigned to index value `i(N-1)`, select `Clip`. When you want to be warned when clipping occurs, select `Clip and warn`. When you want the simulation to stop and the block to display an error when the index values are out of range, select `Error`.

The Scalar Quantizer Encoder block accepts real floating-point and fixed-point inputs. For more information on the data types accepted by each port, see "Data Type Support" on page 2-1329 or "Supported Data Types" on page 2-1335.

Data Type Support

The input data values, boundary points, and codebook values can be input to the block at ports U, B, and C, respectively. The data type of the inputs can be double, single, or Fixed-point.

The outputs of the block can be the index values, the quantized output values, the quantization error, and the clipping status. Use the **Index output data type** parameter to specify the data type of the index output from the block at port I. You can choose int8, uint8, int16, uint16, int32, or uint32. The data type of the quantized output and the quantization error can be double, single, or Fixed-point. The clipping status values output at port S are Boolean values.

Note The input data, boundary points, codebook values, quantized output values, and the quantization error must have the same data type whenever they are present.

Dialog Box

The **Main** pane of the Scalar Quantizer Encoder block dialog appears as follows.

Scalar Quantizer Encoder
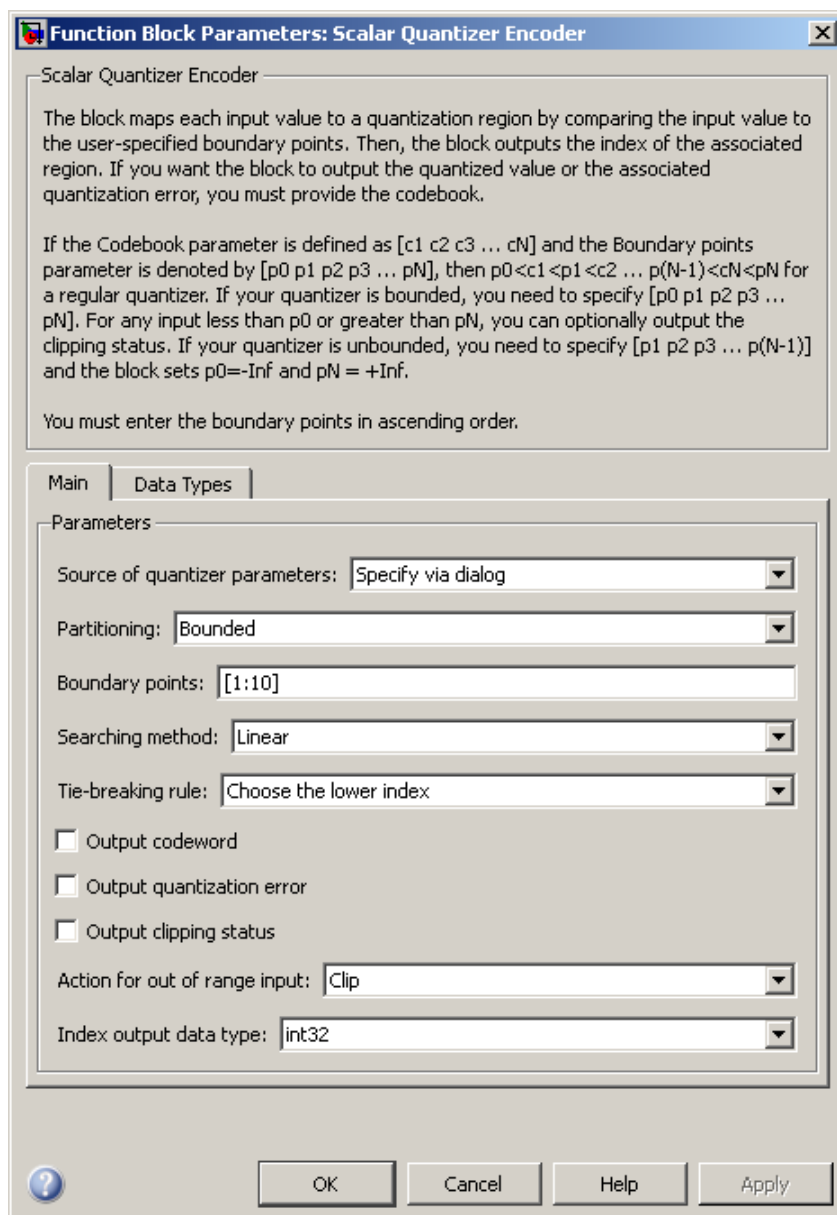
Source of quantizer parameters

Choose `Specify via dialog` to enter the boundary points and codebook values using the block parameters dialog box. Select `Input port` to specify the parameters using the block's input ports. Input the boundary points and codebook values using ports B and C, respectively.

Partitioning

When your quantizer is bounded, select `Bounded`. When your quantizer is unbounded, select `Unbounded`.

Boundary points

Enter a vector of values that represent the boundary points of the quantizer regions. This parameter is visible when you select `Specify via dialog` from the **Source of quantizer parameters** list. Tunable.

Searching method

When you select `Linear`, the block finds the region in which the input value is located using a linear search. When you select `Binary`, the block finds the region in which the input value is located using a binary search.

Tie-breaking rule

Set this parameter to determine the behavior of the block when the input value is the same as the boundary point. When you select `Choose the lower index`, the input value is assigned to lower indexed region. When you select `Choose the higher index`, the value is assigned to the higher indexed region.

Output codeword

Select this check box to output the codeword values that correspond to each index value at port Q(U).

Output quantization error

Select this check box to output the quantization error for each input value at port Err.

Scalar Quantizer Encoder

Codebook

Enter a vector of quantized output values that correspond to each index value. If, for the **Partitioning** parameter, you select Bounded and your boundary points vector has length N, then you must specify a codebook of length N-1. If, for the **Partitioning** parameter, you select Unbounded and your boundary points vector has length N, then you must specify a codebook of length N+1.

This parameter is visible when you select Specify via dialog from the **Source of quantizer parameters** list and you select either the **Output codeword** or **Output quantization error** check box. Tunable.

Output clipping status

When you select this check box, port S appears on the block. Any time an input value is outside the range defined by the **Boundary points** parameter, the block outputs a 1 at this port. When the value is inside the range, the block outputs a 0. This parameter is visible when you select Bounded from the **Partitioning** list.

Action for out of range input

Use this parameter to determine the behavior of the block when an input value is outside the range defined by the **Boundary points** parameter. Suppose the boundary points are defined as [p0 p1 p2 p3 ... pN] and the index values are defined as [i0 i1 i2 ... i(N-1)]. When you want any input value less than p0 to be assigned to index value i0 and any input values greater than pN to be assigned to index value i(N-1), select Clip. When you want to be warned when clipping occurs, select Clip and warn. When you want the simulation to stop and the block to display an error when the index values are out of range, select Error. This parameter is visible when you select Bounded from the **Partitioning** list.

Index output data type

Specify the data type of the index output from the block at port I. You can choose int8, uint8, int16, uint16, int32, or uint32.

The **Data Types** pane of the Scalar Quantizer Encoder block dialog appears as follows.

Scalar Quantizer Encoder
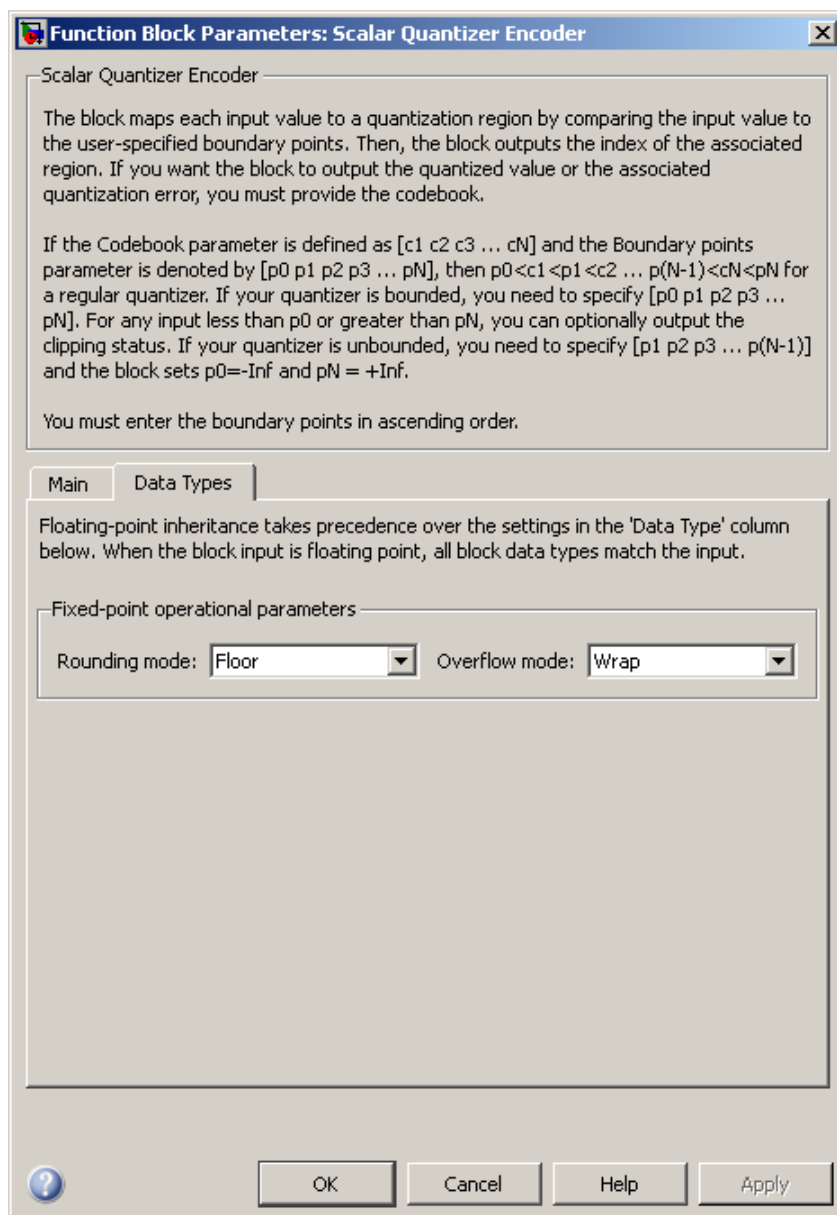
Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode to be used when block inputs are fixed point.

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

Port	Supported Data Types
U	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
B	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
C	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
I	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Scalar Quantizer Encoder

Port	Supported Data Types
Q(U)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Err	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
S	• Boolean

For more information on what data types are supported for each quantizer mode, see "Data Type Support" on page 2-1329.

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Design	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox

Purpose   Select input elements from vector, matrix, or multidimensional signal

Library   Signal Management / Indexing

dspindex

Description   The Selector block is an implementation of the Simulink Selector block. See Selector for more information.

Short-Time FFT

Purpose   Nonparametric estimate of spectrum using short-time, fast Fourier transform (FFT) method

Library   Transforms

dspxfrm3
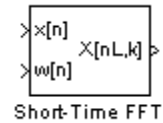
Description   The Short-Time FFT block computes a nonparametric estimate of the spectrum. The block buffers, applies a window, and zero pads the input signal. The block then takes the FFT of the signal, transforming it into the frequency domain.

Short-Time FFT

Connect your single-channel analysis window to the w(n) port. For the **Analysis window length** parameter, enter the length of the analysis window, *W*. The block buffers the input signal such that it has a frame length of *W*

Connect your single-channel or multichannel input signal to the x(n) port. After the block buffers and windows this signal, it zero-pads the signal before computing the FFT. For the **FFT length** parameter, enter the length to which the block pads the input signal. For the **Overlap between consecutive windows (in samples)** parameter, enter the number of samples to overlap each frame of the input signal.

The block outputs the complex-valued, single-channel or multichannel short-time FFT at port X(n,k).

Fixed-Point Data Types

The following diagram shows the data types used within the Short-Time FFT subsystem block for fixed-point signals.
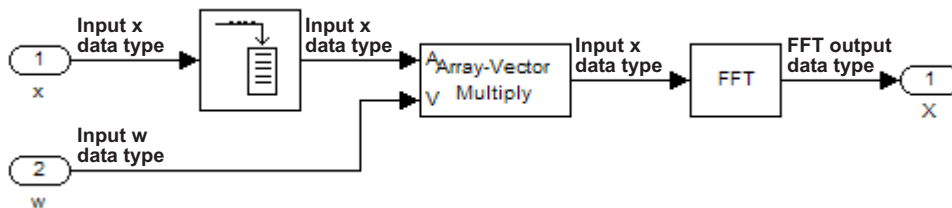
The settings for the fixed-point parameters of the Array-Vector Multiply block in the diagram above are as follows:

- **Rounding Mode** — Floor
- **Overflow Mode** — Wrap
- **Product output** — Inherit via internal rule
- **Accumulator** — Inherit via internal rule
- **Output** — Same as first input

The settings for the fixed-point parameters of the FFT block in the diagram above are as follows:

- **Rounding Mode** — Floor
- **Overflow Mode** — Wrap
- **Sine table** — Same word length as input
- **Product output** — Inherit via internal rule
- **Accumulator** — Inherit via internal rule
- **Output** — Inherit via internal rule

See the FFT and Array-Vector Multiply block reference pages for more information.

Examples   The dspstsa_win32 example illustrates how to use the Short-Time FFT and Inverse Short-Time FFT blocks to remove the background noise from a speech signal.

Short-Time FFT
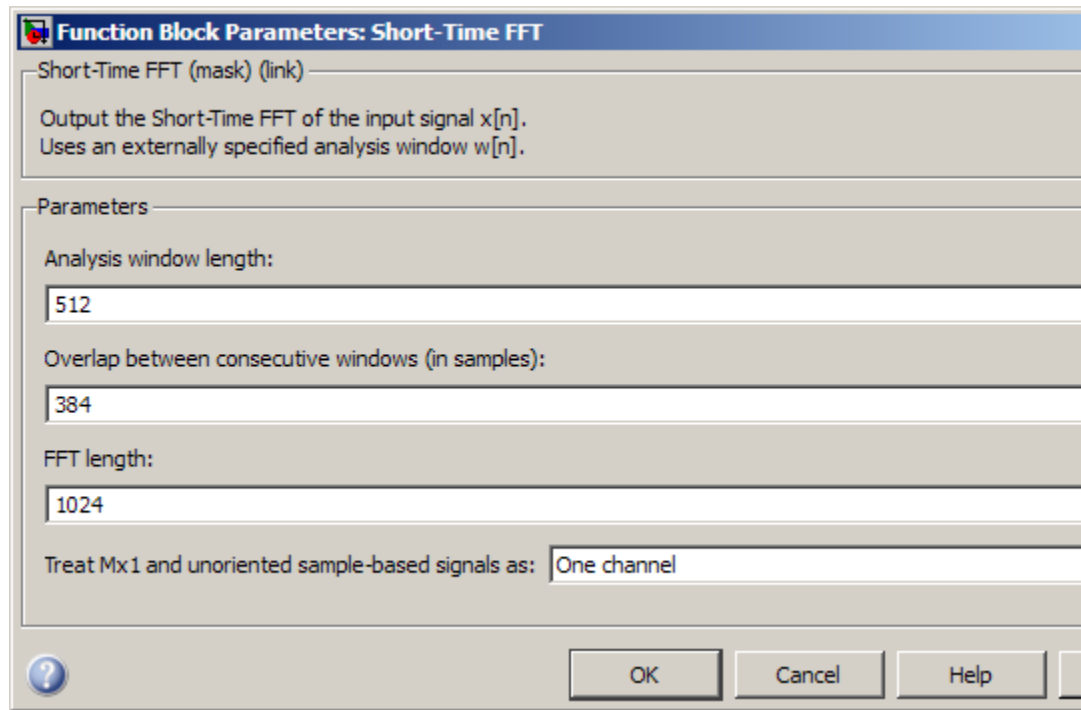
**Dialog
Box**

Analysis window length

Specify the frame length of the analysis window. The **Analysis
window length** must be a positive integer value greater than
one.

Overlap between consecutive windows (in samples)

Enter the number of samples of overlap for each frame of the
input signal.

FFT length

Enter the length to which the block pads the input signal.

Treat Mx1 and unoriented sample-based signals as
Specify how the block treats sample-based *M*-by-1 column vectors and unoriented sample-based vectors of length *M*. You can select one of the following options:

- One channel — When you select this option, the block treats *M*-by-1 and unoriented sample-based inputs as a column vector (one channel).

- M channels (this choice will be removed see release notes) — When you select this option, the block treats *M*-by-1 and unoriented sample-based inputs as a 1-by-*M* row vector.

Note This parameter will be removed in a future release. See the *DSP System Toolbox Release Notes* for more information.

References   Quatieri, Thomas E. *Discrete-Time Speech Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 2001.

Supported Data Types

Port	Supported Data Types
x(n)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
w(n)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only)

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers
X(n,k)	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

See Also

Burg Method	DSP System Toolbox
Inverse Short-Time FFT	DSP System Toolbox
Magnitude FFT	DSP System Toolbox
Periodogram	DSP System Toolbox
Spectrum Scope	DSP System Toolbox
Window Function	DSP System Toolbox
Yule-Walker Method	DSP System Toolbox
pwelch	Signal Processing Toolbox

See "Spectral Analysis" for related information.

Signal From Workspace

Purpose   Import signal from MATLAB workspace

Library   Sources

dspsrcs4

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.

1:10

When the **Signal** parameter specifies an M-by-N matrix (M≠1), each
of the N columns is treated as a distinct channel. You specify the
frame size in the **Samples per frame** parameter, M_o, and the output
is an M_o-by-N matrix containing M_o consecutive samples from each
signal channel. You specify the output sample period in the **Sample
time** parameter, T_s, and the output frame period is M_o*T_s. For $M_o=1$,
the output is sample based; otherwise the output is frame based. For
convenience, an imported row vector (M=1) is treated as a single
channel, so the output dimension is M_o-by-1.

When the **Signal** parameter specifies an M-by-N-by-P array, each of
the P pages (an M-by-N matrix) is output in sequence with period T_s.
The **Samples per frame** parameter must be set to 1, and the output is
always sample based.

Initial and Final Conditions

Unlike the Simulink From Workspace block, the Signal From
Workspace block holds the output value constant between successive
output frames (that is, no linear interpolation takes place). Additionally,
the initial signal values are always produced immediately at *t*=0.

When the block has output all of the available signal samples, it can
start again at the beginning of the signal, or simply repeat the final
value or generate zeros until the end of the simulation. (The block does
not extrapolate the imported signal beyond the last sample.) The **Form
output after final data value by** parameter controls this behavior:

- When you specify `Setting To Zero`, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.

- When you specify `Holding Final Value`, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.

- When you specify `Cyclic Repetition`, the block repeats the signal from the beginning after it reaches the last sample in the signal. If the frame size you specify in the **Samples per frame** parameter does not evenly divide the input length, a buffer block is inserted into the Signal From Workspace subsystem, and the model becomes multirate. If you do not want your model to become multirate, make sure the frame size evenly divides the input signal length.

  Select the **Warn when frame size does not evenly divide input length** parameter to be alerted when the input length is not an integer multiple of the frame size and your model will become multirate. Use the Model Explorer to turn these warnings on or off model-wide:

  1 Select **Model Explorer** from the **View** menu in your model.

  2 In the **Search** bar of the Model Explorer, search by `Property Name` for the `ignoreOrWarnInputAndFrameLengths` property. Each block with the **Warn when frame size does not evenly divide input length** check box appears in the list in the **Contents** pane.

  3 Select each of the blocks for which you wish to toggle the warning parameter, and select or deselect the check box in the `ignoreOrWarnInputAndFrameLengths` column.

Examples

Example 1

In the first model, ex_signal_from_workspace_fb, 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

Signal From Workspace

specifies Setting To Zero, so all outputs after the third frame (at $t=8$) are zero.
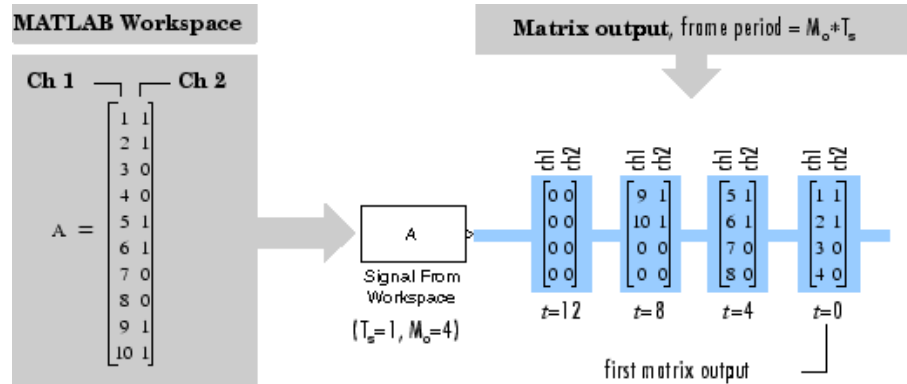
Example 2

In the second model, ex_signal_from_workspace_sb, 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.
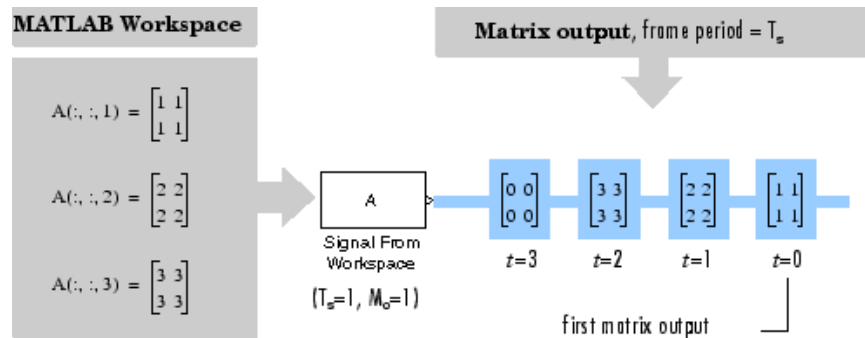
The **Samples per frame** parameter is set to 1 for 3-D input.

**Dialog
Box**
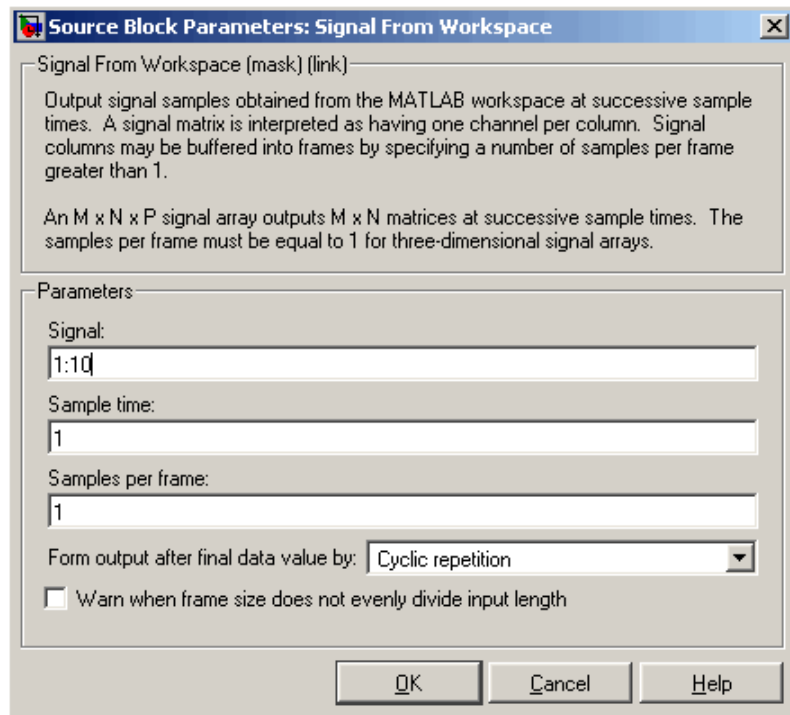
Signal

> 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 when you specify a 3-D array in the **Signal** parameter.

Signal From Workspace

Form output after final data value by

Specifies the output after all of the specified signal samples have been generated. The block can output zeros for the duration of the simulation (`Setting to zero`), repeat the final data sample (`Holding Final Value`) or repeat the entire signal from the beginning (`Cyclic Repetition`).

Warn when frame size does not evenly divide input length

Select this parameter to be alerted when the input length is not an integer multiple of the frame size and your model will become multirate. For more information, see "Initial and Final Conditions" on page 2-1344.

This parameter is only visible when `Cyclic Repetition` is selected for the **Form output after final data value by** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

From Audio Device	DSP System Toolbox
From Wave File (Obsolete)	DSP System Toolbox
Signal From Workspace	DSP System Toolbox
From Workspace	Simulink
To Workspace	Simulink
Triggered Signal From Workspace	DSP System Toolbox

See the sections below for related information:

- "Create Sample-Based Signals"
- "Create Frame-Based Signals"
- "Import and Export Sample-Based Signals"
- "Import and Export Frame-Based Signals"

Signal To Workspace

Purpose   Write data to MATLAB workspace

Library   Sinks

              dspsnks4

Description
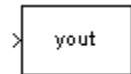
The Signal To Workspace block writes data from your simulation into an array or structure in the main MATLAB workspace. You can specify a name for the workspace variable as well as whether the data is saved as an array, structure, or structure with time.

When the **Save format** is set to Array or Structure, the dimensions of the output depend on the input dimensions and the setting of the **Save 2-D signals as** parameter. The following table summarizes the output dimensions under various conditions. In the table, *K* represents the value of the **Limit data points to last** parameter.

Input Signal Dimensions	Save 2-D Signals as ...	Signal To Workspace Output Dimension
M-by-*N* matrix	2-D array (concatenate along first dimension)	*K*-by-*N* matrix. If you set the **Limit data points to last** parameter to inf, *K* represents the total number of samples acquired in each column by the end of simulation. This is equivalent to multiplying the input frame size (*M*) by the total number

Input Signal Dimensions	Save 2-D Signals as ...	Signal To Workspace Output Dimension
		of *M*-by-*N* inputs acquired by the block.
M-by-*N* matrix	`3-D array (concatenate along third dimension)`	*M*-by-*N*-by-*K* array. If you set the **Limit data points to last** parameter to `inf`, *K* represents the total number of *M*-by-*N* inputs acquired by the end of the simulation.
Length-*N* unoriented vector	Any setting	*K*-by-*N* matrix
N-dimensional array where *N* > 2	Any setting	Array with *N+1* dimensions, where the size of the last dimension is equal to *K*. If you set the **Limit data points to last** parameter to `inf`, *K* represents the total number of *M*-by-*N* inputs acquired by the end of simulation

Examples

Example 1: Save 2-D Signals as a 2-D Array

In the ex_signaltoworkspace_ref2 model, the Signal To Workspace block receives a 2-by-4 matrix input and logs 11 frames (two samples per frame) by the end of the simulation. Because the **Save 2-D signals as** parameter is set to 2-D array (concatenate along first

Signal To Workspace

dimension), the block concatenates the input along the first dimension to create a 22-by-4 matrix, *A*, in the MATLAB workspace.

The following figure illustrates the behavior of the Signal to Workspace block in this example.
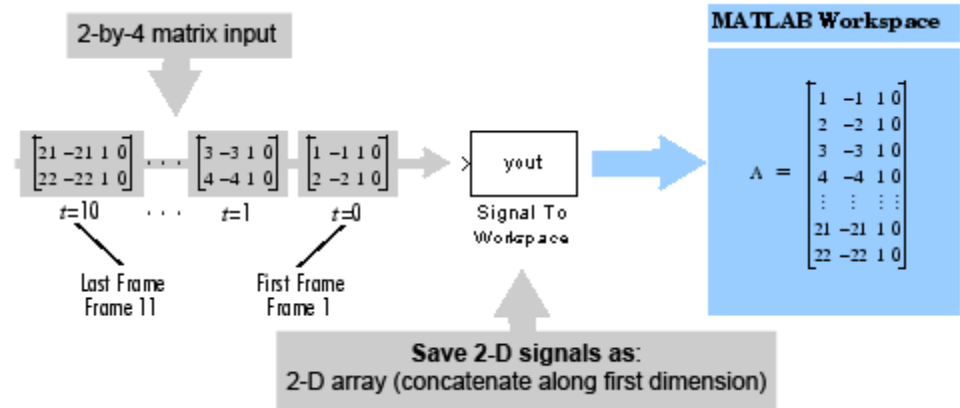
In the 2-D output mode, there is no indication of where one frame ends and another begins. To log input frames separately, set the **Save 2-D signals as** parameter to 3-D array (concatenate along third dimension), as shown in Example 2.

Example 2: Save 2-D Signals as a 3-D Array

In the ex_signaltoworkspace_ref1 model, the input to the Signal To Workspace block is a 2-by-4 matrix. The **Save 2-D signals as** parameter is set to 3-D array (concatenate along third dimension), so by the end of the simulation the Signal To Workspace block logs 11 frames of data as a 2-by-4-by-11 array, *A*, in the MATLAB workspace.

The following figure illustrates the behavior of the Signal to Workspace block in this example.

Signal To Workspace
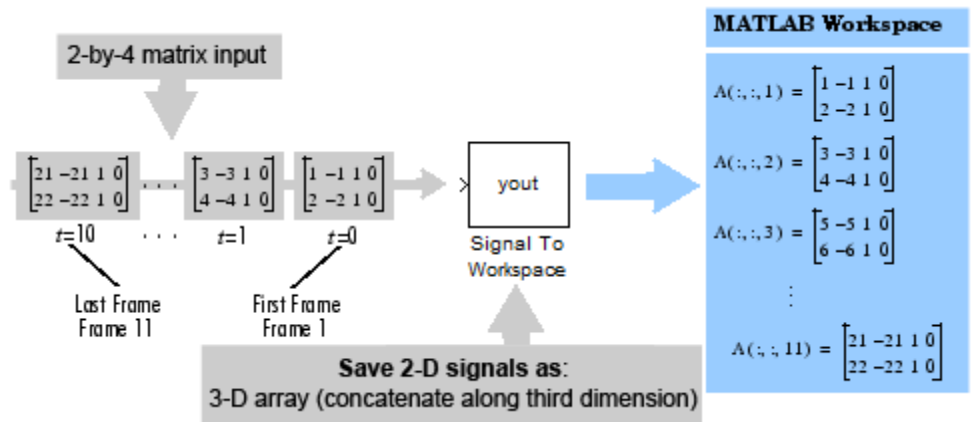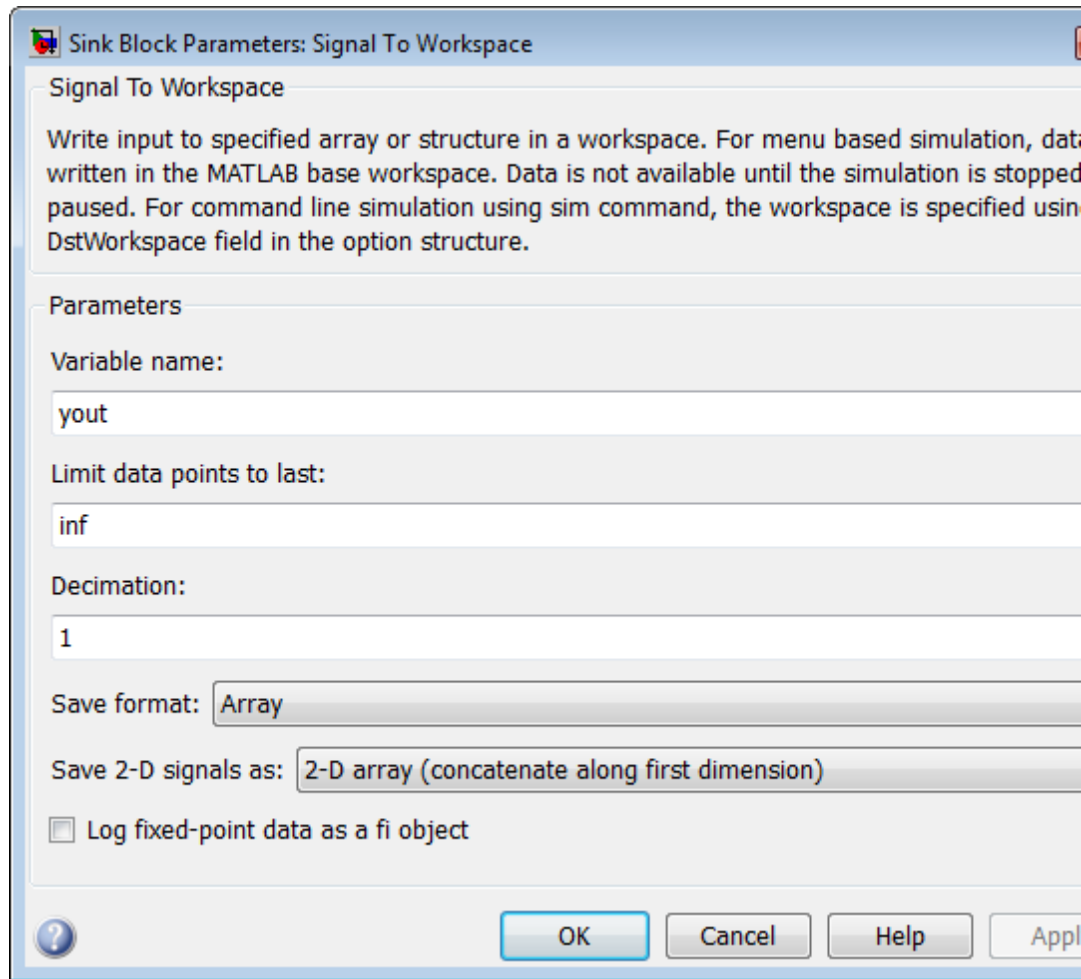
Variable name

Specify the name of the array or structure into which the block logs the simulation data. The block creates this variable in the MATLAB workspace only after the simulation stops running.

When you enter the name of an existing workspace variable, the block overwrites that variable with the simulation data.

Limit data points to last

Specify the maximum number of samples or frames the block will save. When the simulation generates more than the specified maximum number of samples or frames, the simulation saves only the most recently generated data. To capture all data, set this parameter to `inf`. See the table in the Description section for more information on how this parameter affects the dimensions of the logged data.

Decimation

Specify a positive integer d to determine how often the block writes data to the workspace array or structure. The block writes data to the array or structure every dth sample. With the default decimation value of `1`, the block writes data at every time step.

Save format

Specify the format in which to save simulation output to the workspace. You can select one of the following options:

- `Array` — Select this option to save the data as an N-dimensional array. If the input signal is an unoriented vector, the resulting workspace array is 2-D. Each input vector is saved in a row of the output matrix, vertically concatenated onto the previous vector. If the input signal is 2-dimensional, the dimensions of the resulting workspace array depend on the setting of the **Save 2-D signals as** parameter.

- `Structure` — Select this option to save the data as a structure consisting of three fields: `time`, `signals` and `blockName`. In this mode, the `time` field is empty, and the `blockName` field contains the name of the Signal To Workspace block. The `signals` field contains a structure with three additional fields: `values`, `dimensions`, and `label`. The `values` field contains the array of signal values, the `dimensions` field specifies the dimensions of the values array, and the `label` field contains the label of the input line.

- `Structure with time` — This option is the same as `Structure`, except that the `time` field contains a vector of simulation time steps. This is the only output format that can be read directly by a From Workspace block. When you select this option, the **Save 2-D signals as** parameter is not available. In this mode, the block always saves 2-D input arrays as a 3-D array.

The default setting of this parameter is `Array`.

Save 2-D signals as

Specify whether the block outputs 2-D signals as a 2-D or 3-D array in the MATLAB workspace:

- `2-D array (concatenate along first dimension)` — When you select this option, the block saves an *M*-by-*N* input signal as a (*K*M*)-by-*N* matrix, where *K*M* is the total number of samples acquired by the end of the simulation. The block vertically concatenates each *M*-by-*N* matrix input with the previous input to produce the 2-D output array. See "Example 1: Save 2-D Signals as a 2-D Array" on page 2-1351 for more information about this mode.

- `3-D array (concatenate along third dimension)` — When you select this option, the block saves an *M*-by-*N* input signal as an *M*-by-*N*-by-*K* array, where *K* is the number of *M*-by-*N* inputs logged by the end of the simulation. *K* has an upper bound equal to the value of the **Limit data points to last** parameter. See "Example 2: Save 2-D Signals as a 3-D Array" on page 2-1352 for more information about this mode.

This parameter is visible only when you set the **Save format** parameter to `Array` or `Structure`. When you set the **Save format** parameter to `Structure with time`, the block outputs the 2-D input signal as a 3-D array.

> **Note** The Inherit from input (this choice will be removed - see release notes) option will be removed in a future release. See "Signal To Workspace Block Changes" in the *DSP System Toolbox Release Notes* for more information.

Log fixed-point data as a fi object
> Select this check box to log fixed-point data to the MATLAB workspace as a Fixed-Point Toolbox fi object. Otherwise, fixed-point data is logged to the workspace as double.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

Triggered To Workspace	DSP System Toolbox
To Workspace	Simulink

Sine Wave

Purpose   Generate continuous or discrete sine wave

Library   Sources

dspsrcs4

Description   The Sine Wave block generates a multichannel real or complex sinusoidal signal, with independent amplitude, frequency, and phase in each output channel. A real sinusoidal signal is generated when the **Output complexity** parameter is set to Real, and is defined by an expression of the type

$$y = A\sin(2\pi ft + \phi)$$

where you specify A in the **Amplitude** parameter, f in hertz in the **Frequency** parameter, and φ in radians in the **Phase offset** parameter. A complex exponential signal is generated when the **Output complexity** parameter is set to Complex, and is defined by an expression of the type

$$y = Ae^{j(2\pi ft + \phi)} = A\{\cos(2\pi ft + \phi) + j\sin(2\pi ft + \phi)\}$$

Sections of This Reference Page

- "Generating Multichannel Outputs" on page 2-1359
- "Output Sample Time and Samples Per Frame" on page 2-1359
- "Sample Mode" on page 2-1359
- "Discrete Computational Methods" on page 2-1360
- "Examples" on page 2-1363
- "Dialog Box" on page 2-1364
- "Supported Data Types" on page 2-1369
- "See Also" on page 2-1369

Generating Multichannel Outputs

For both real and complex sinusoids, the **Amplitude**, **Frequency**, and **Phase offset** parameter values (*A*, *f*, and *φ*) can be scalars or length-*N* vectors, where *N* is the desired number of channels in the output. When you specify at least one of these parameters as a length-*N* vector, scalar values specified for the other parameters are applied to every channel.

For example, to generate the three-channel output containing the real sinusoids below, set **Output complexity** to Real and the other parameters as follows:

- **Amplitude** = [1 2 3]
- **Frequency** = [1000 500 250]
- **Phase offset** = [0 0 pi/2]

$$
y = \begin{cases} \sin(2000\pi t) & \text{(channel 1)} \\ 2\sin(1000\pi t) & \text{(channel 2)} \\ 3\sin\left(500\pi t + \dfrac{\pi}{2}\right) & \text{(channel 3)} \end{cases}
$$

Output Sample Time and Samples Per Frame

In all discrete modes, the block buffers the sampled sinusoids into frames of size M, where you specify M in the **Samples per frame** parameter. The output is a frame-based M-by-N matrix with frame period M*T_s, where you specify T_s in the **Sample time** parameter. For M=1, the output is sample based.

Sample Mode

The **Sample mode** parameter specifies the block's sampling property, which can be Continuous or Discrete:

- Continuous

Sine Wave

In continuous mode, the sinusoid in the *i*th channel, y_i, is computed as a continuous function,

$$y_i = A_i \sin\left(2\pi f_i t + \phi_i\right) \quad \text{(real)}$$

or

$$y_i = A_i e^{j\left(2\pi f_i t + \phi_i\right)} \qquad \text{(complex)}$$

and the block's output is continuous. In this mode, the block's operation is the same as that of a Simulink Sine Wave block with **Sample time** set to 0. This mode offers high accuracy, but requires trigonometric function evaluations at each simulation step, which is computationally expensive. Additionally, because this method tracks absolute simulation time, a discontinuity will eventually occur when the time value reaches its maximum limit.

Note also that many DSP System Toolbox blocks do not accept continuous-time inputs.

- Discrete

  In discrete mode, the block's discrete-time output can be generated by directly evaluating the trigonometric function, by table lookup, or by a differential method. The three options are explained below.

Discrete Computational Methods

When you select Discrete from the **Sample mode** parameter, the secondary **Computation method** parameter provides three options for generating the discrete sinusoid:

- Trigonometric Fcn

- Table Lookup

- Differential

Note To generate fixed-point sinusoids, you must select Table Lookup.

Trigonometric Fcn

The trigonometric function method computes the sinusoid in the ith channel, y_i, by sampling the continuous function

$$y_i = A_i \sin\left(2\pi f_i t + \phi_i\right) \quad (\text{real})$$

or

$$y_i = A_i e^{j\left(2\pi f_i t + \phi_i\right)} \qquad (\text{complex})$$

with a period of T_s, where you specify T_s in the **Sample time** parameter. This mode of operation shares the same benefits and liabilities as the Continuous sample mode described above.

At each sample time, the block evaluates the sine function at the appropriate time value *within the first cycle* of the sinusoid. By constraining trigonometric evaluations to the first cycle of each sinusoid, the block avoids the imprecision of computing the sine of very large numbers, and eliminates the possibility of discontinuity during extended operations (when an absolute time variable might overflow). This method therefore avoids the memory demands of the table lookup method at the expense of many more floating-point operations.

Table Lookup

The table lookup method precomputes the *unique* samples of every output sinusoid at the start of the simulation, and recalls the samples from memory as needed. Because a table of finite length can only be constructed when all output sequences repeat, the method requires that the period of every sinusoid in the output be evenly divisible by the sample period. That is, $1/(f_i T_s) = k_i$ must be an integer value for every channel i = 1, 2, ..., N.

When the **Optimize table for** parameter is set to Speed, the table constructed for each channel contains k_i elements. When the **Optimize table for** parameter is set to Memory, the table constructed for each channel contains $k_i/4$ elements.

For long output sequences, the table lookup method requires far fewer floating-point operations than any of the other methods, but can demand considerably more memory, especially for high sample rates (long tables). This is the recommended method for models that are intended to emulate or generate code for DSP hardware, and that therefore need to be optimized for execution speed.

Note The lookup table for this block is constructed from double-precision floating-point values. Thus, when you use the Table lookup computation mode, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** or **User-defined** data type to values greater than 53 bits does not improve the precision of your output.

Differential

The differential method uses an incremental algorithm. This algorithm computes the output samples based on the output values computed at the previous sample time (and precomputed update terms) by making use of the following identities.

$$\sin(t + T_s) = \sin(t)\cos(T_s) + \cos(t)\sin(T_s)$$
$$\cos(t + T_s) = \cos(t)\cos(T_s) - \sin(t)\sin(T_s)$$

The update equations for the sinusoid in the ith channel, y_i, can therefore be written in matrix form as

$$\begin{bmatrix} \sin\{2\pi f_i(t+T_s)+\phi_i\} \\ \cos\{2\pi f_i(t+T_s)+\phi_i\} \end{bmatrix} = \begin{bmatrix} \cos(2\pi f_i T_s) & \sin(2\pi f_i T_s) \\ -\sin(2\pi f_i T_s) & \cos(2\pi f_i T_s) \end{bmatrix} \begin{bmatrix} \sin(2\pi f_i t+\phi_i) \\ \cos(2\pi f_i t+\phi_i) \end{bmatrix}$$

where you specify T_s in the **Sample time** parameter. Since T_s is constant, the right-hand matrix is a constant and can be computed once at the start of the simulation. The value of $A_i\sin[2\pi f_i(t+T_s)+\varphi_i]$ is then computed from the values of $\sin(2\pi f_i t+\varphi_i)$ and $\cos(2\pi f_i t+\varphi_i)$ 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 example provides a comparison of all the available sine generation methods.

Sine Wave

Dialog Box

The **Main** pane of the Sine Wave block dialog appears as follows.

Sine Wave

Dialog Box

The **Main** pane of the Sine Wave block dialog appears as follows.

Source Block Parameters: Sine Wave

Sine Wave (mask) (link)

Output samples of a sinusoid. To generate more than one sinusoid simultaneously, enter a vector of values for the Amplitude, Frequency, and Phase offset parameters.

Main | Data Types

Amplitude:

1

Frequency (Hz):

100

Phase offset (rad):

0

Sample mode: Discrete

Output complexity: Real

Computation method: Trigonometric fcn

Sample time:

1/1000

Samples per frame:

1

Resetting states when re-enabled: Restart at time zero

OK   Cancel   Help

Amplitude

A length-*N* vector containing the amplitudes of the sine waves in each of *N* output channels, or a scalar to be applied to all *N* channels. The vector length must be the same as that specified for the **Frequency** and **Phase offset** parameters. Tunable when **Computation method** is to `Trigonometric fcn` or `Differential`.

Frequency

A length-*N* vector containing frequencies, in Hertz, of the sine waves in each of *N* output channels, or a scalar to be applied to all *N* channels. The vector length must be the same as that specified for the **Amplitude** and **Phase offset** parameters. You can specify positive, zero, or negative frequencies. Tunable when **Sample mode** is `Continuous` or **Computation method** is `Trigonometric fcn`.

Phase offset

A length-*N* vector containing the phase offsets, in radians, of the sine waves in each of *N* output channels, or a scalar to be applied to all *N* channels. The vector length must be the same as that specified for the **Amplitude** and **Frequency** parameters. Tunable when **Sample mode** is `Continuous` or **Computation method** is `Trigonometric fcn`.

Sample mode

The block's sampling behavior, `Continuous` or `Discrete`. This parameter is not tunable.

Output complexity

The type of waveform to generate: `Real` specifies a real sine wave, `Complex` specifies a complex exponential. This parameter is not tunable.

Computation method

The method by which discrete-time sinusoids are generated: `Trigonometric fcn`, `Table lookup`, or `Differential`. This parameter is not tunable. For more information on each of the

available options, see "Discrete Computational Methods" on page 2-1360 in the Description section.

This parameter is only visible when you set the **Sample mode** to Discrete.

Note To generate fixed-point sinusoids, you must set the **Computation method** to Table lookup.

Optimize table for

Optimizes the table of sine values for Speed or Memory (this parameter is only visible when the **Computation method** parameter is set to Table lookup). When optimized for speed, the table contains k elements, and when optimized for memory, the table contains $k/4$ elements, where k is the number of input samples in one full period of the sine wave.

Sample time

The period with which the sine wave is sampled, T_s. The block's output frame period is $M*T_s$, where you specify M in the **Samples per frame** parameter. This parameter is disabled when you select Continuous from the **Sample mode** parameter. This parameter is not tunable.

Samples per frame

The number of consecutive samples from each sinusoid to buffer into the output frame, M. When the value of this parameter is 1, the block outputs a sample-based signal.

This parameter is disabled when you select Continuous from the **Sample mode** parameter.

Resetting states when re-enabled

This parameter only applies when the Sine Wave block is located inside an enabled subsystem and the **States when enabling** parameter of the Enable block is set to reset. This parameter

determines the behavior of the Sine Wave block when the subsystem is re-enabled. The block can either reset itself to its starting state (`Restart at time zero`), or resume generating the sinusoid based on the current simulation time (`Catch up to simulation time`). This parameter is disabled when you select `Continuous` from the **Sample mode** parameter.

The **Data Types** pane of the Sine Wave block dialog appears as follows.
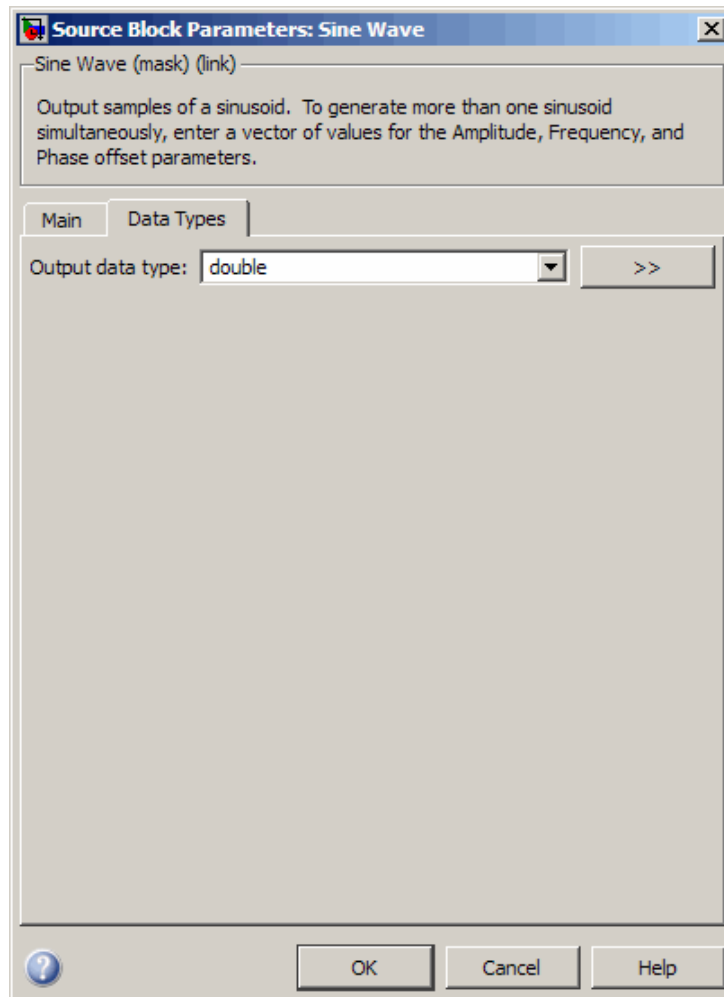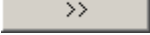
Sine Wave

Output data type

> Specify the output data type for this block. You can select one of the following:

- A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`. When you select this option, the output data type and scaling matches that of the next downstream block.

- A built in data type, such as `double`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Note The lookup table for this block is constructed from double-precision floating-point values. Thus, when you use the `Table lookup` computation mode, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length of the **Output** or **User-defined** data type to values greater than 53 bits does not improve the precision of your output.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

Chirp	DSP System Toolbox
Complex Exponential	DSP System Toolbox

Sine Wave

Signal From Workspace	DSP System Toolbox
Signal Generator	Simulink
Sine Wave	Simulink
sin	MATLAB

Purpose   Factor matrix using singular value decomposition

Library   Math Functions / Matrices and Linear Algebra / Matrix Factorizations

dspfactors
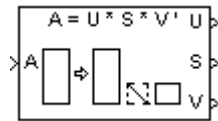
Description   The Singular Value Decomposition block factors the M-by-N input
matrix A such that

$$A = U \cdot diag(S) \cdot V^*$$

where

- U is an M-by-P matrix
- V is an N-by-P matrix
- S is a length-P vector
- P is defined as $\min(M,N)$

When

- $M = N$, U and V are both M-by-M unitary matrices
- $M > N$, 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
- $N > M$, U is an M-by-M unitary matrix, and V is an N-by-M matrix
  whose columns are the first M columns of a unitary matrix

In all cases, S is an unoriented vector of positive singular values having
length P.

Length-N row inputs are treated as length-N columns.

Note that the first (maximum) element of output S is equal to the
2-norm of the matrix A.

Singular Value Decomposition

Dialog Box
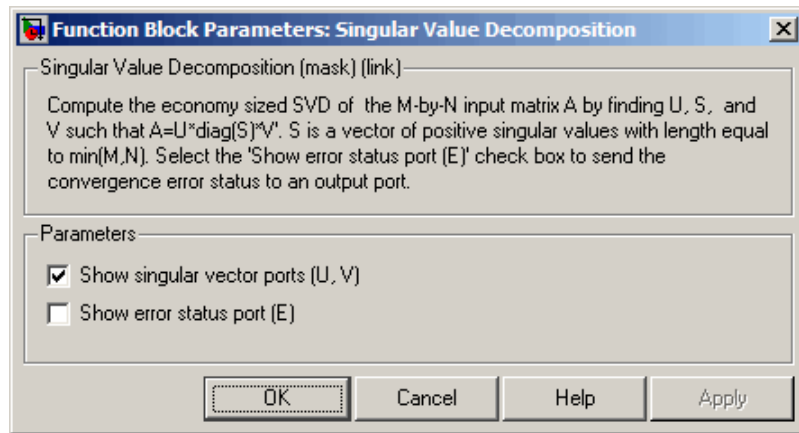
Show singular vector ports
Select to enable the U and V output ports.

Show error status port
Select to enable the E output port, which reports a failure to converge. The possible values you can receive on the port are:

- 0 — The singular value decomposition calculation converges.

- 1 — The singular value decomposition calculation does not converge.

If the singular value decomposition calculation fails to converge, the output at ports U, S, and V are undefined matrices of the correct size.

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point
U	• Double-precision floating point • Single-precision floating point
S	• Double-precision floating point • Single-precision floating point
V	• Double-precision floating point • Single-precision floating point
E	• Boolean

See Also

Autocorrelation LPC	DSP System Toolbox
Cholesky Factorization	DSP System Toolbox
LDL Factorization	DSP System Toolbox
LU Inverse	DSP System Toolbox
Pseudoinverse	DSP System Toolbox
QR Factorization	DSP System Toolbox
SVD Solver	DSP System Toolbox
svd	MATLAB

See "Matrix Factorizations" for related information.

Sort

Purpose	Sort input elements by value

Library   Statistics

dspstat3

Description
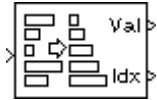
The Sort block ranks the values of the input elements using either a quick sort or an insertion sort algorithm. The quick sort algorithm uses a recursive sort method and is faster at sorting more than 32 elements. The insertion sort algorithm uses a non-recursive method and is faster at sorting less than 32 elements. You should also always use the insertion sort algorithm when you are generating code from the Sort block if you do not want recursive function calls in your code. To specify the sort method, use the **Sort algorithm** parameter.

The **Mode** parameter specifies the block's mode of operation, and can be set to Value, Index, or Value and index.

Value Mode

When **Mode** is set to Value, the block sorts the elements in each column of the *M*-by-*N* input matrix u in order of ascending or descending value, as specified by the **Sort order** parameter.

```
val = sort(u)
val = flipud(sort(u))
```

The output at each sample time, val, is an *M*-by-*N* matrix containing the sorted columns of u.

The block sorts complex inputs according to their magnitude.

Index Mode

When **Mode** is set to Index, the block sorts the elements in each column of the *M*-by-*N* input matrix u,

```
[val,idx] = sort(u)
[val,idx] = flipud(sort(u))
```

and outputs the *M*-by-*N* index matrix, `idx`. The jth column of `idx` is an index vector that permutes the jth column of `u` to the desired sorting order.

```
val(:,j) = u(idx(:,j),j)
```

The index value outputs are always 32-bit unsigned integer values.

Value and Index Mode

When **Mode** is set to `Value and index`, the block outputs both the sorted matrix, `val`, and the index matrix, `idx`.

Fixed-Point Data Types

The parameters on the **Data Types** pane are only used for complex fixed-point inputs. Complex fixed-point inputs are sorted by magnitude squared. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 2-1374. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Sort

Dialog Box

The **Main** pane of the Sort block dialog appears as follows.
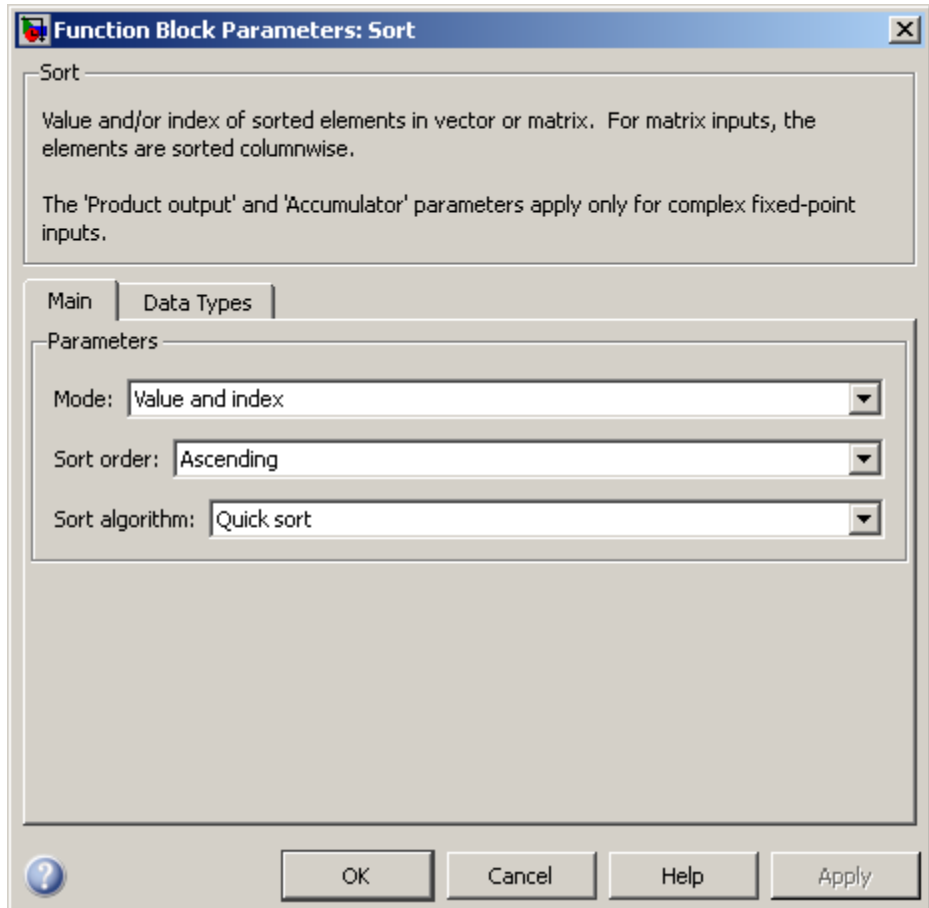
Mode

Specify the block's mode of operation: Output the sorted matrix (Value), the index matrix (Index), or both (Value and index).

Sort order

> Specify the order in which to sort the training points, `Descending` or `Ascending`.

Sort algorithm

> Specify whether the elements of the input are sorted using a `Quick sort` or an `Insertion sort` algorithm.

The **Data Types** pane of the Sort block dialog appears as follows.

Sort
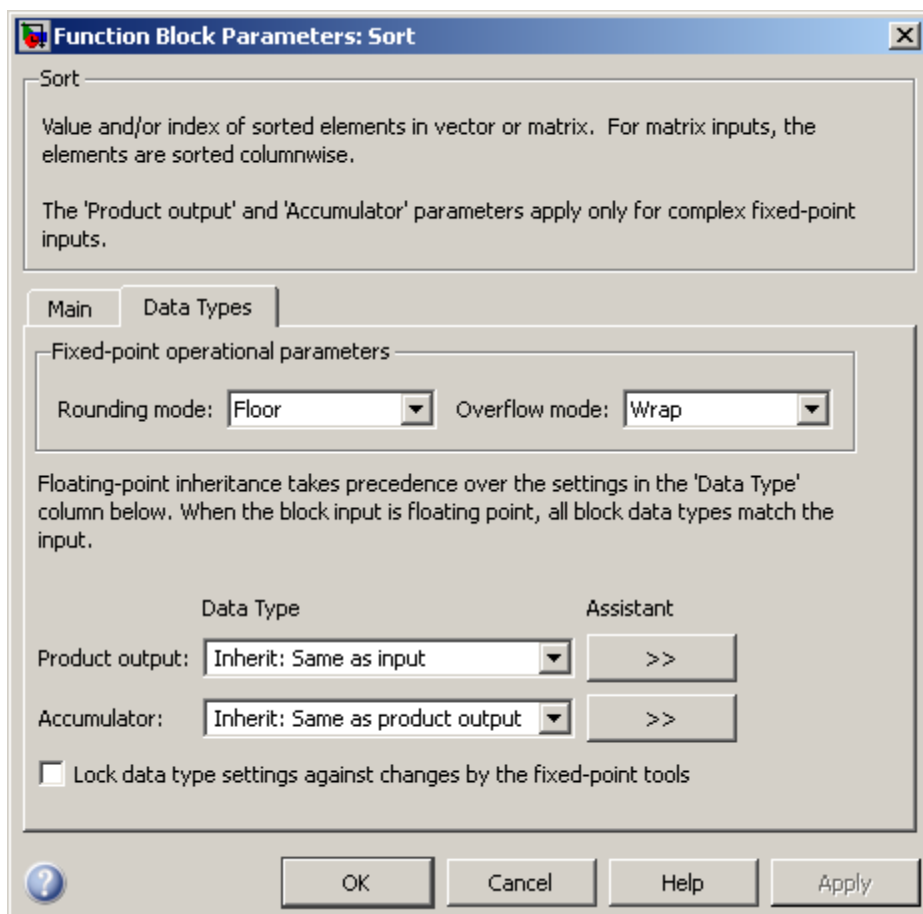
> **Note** The parameters on the **Data Types** pane are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 2-1374. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Rounding mode
>   Select the rounding mode for fixed-point operations.
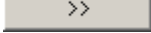
Overflow mode
>   Select the overflow mode for fixed-point operations.

Product output data type
>   Specify the product output data type. See "Fixed-Point Data Types" on page 2-1375 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:
>
>   - A rule that inherits a data type, for example, Inherit: Same as input
>
>   - An expression that evaluates to a valid data type, for example, fixdt([],16,0)
>
>   Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.
>
>   See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Sort
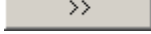
Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-1375 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Same as product output`

- An expression that evaluates to a valid data type, for example, `fixdt([],16,0)`

Click the **Show data type assistant** button $\boxed{\quad \gg \quad}$ to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, 32-, and 128-bit unsigned integers
	• 8-, 16-, 32-, and 128-bit signed integers
Val	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, 32-, and 128-bit unsigned integers

Port	Supported Data Types
	• 8-, 16-, 32-, and 128-bit signed integers
	• 32-bit unsigned integers

See Also

Histogram	DSP System Toolbox
Idx	
Median	DSP System Toolbox
sort	MATLAB

Spectrum Scope

Purpose   Display mean-square spectrum or power spectral density of each input signal

Library   Sinks

dspsnks4

Description   The Spectrum Scope block computes and displays the mean-square spectrum or power spectral density of each input signal. The input can be a vector or a matrix.

Note When the **Buffer input** and **Specify FFT length** parameters are both cleared, the block input length must be a power of two.

For an example that uses the Spectrum Scope block, see "Display Frequency-Domain Data" section in the DSP System Toolbox documentation.

Note For information about the Spectrum Analyzer system object, see dsp.SpectrumAnalyzer.

Scope Properties Pane

The **Spectrum units** parameter allows you to specify the following information:

- The type of measurement for the block to compute (Power Spectral Density or Mean-Square Spectrum)

- The type of scaling for the block to use (linear or log)

You can set the **Spectrum units** parameter to one of the options shown in the following table.

Spectrum Units	Measurement Type	Scaling
`Watts`	Mean-Square Spectrum (MSS)	Linear
`dBW` `dBm`	Mean-Square Spectrum (MSS)	Logarithmic
`Watts/Hertz`	Power Spectral Density (PSD)	Linear
`dBW/Hertz` `dBm/Hertz`	Power Spectral Density (PSD)	Logarithmic

The *X*-axis units are always expressed in Hertz. The spacing between frequency points is $1/(N_{fft}T_s)$.

The **Spectrum type** parameter specifies the range of frequencies over which the block computes the spectrum. The available options are `One-sided ([0...Fs/2])` and `Two-sided ((-Fs/2...Fs/2])`, where F_s is the sampling frequency of the original time-domain signal. Both the one-sided and two-sided options compute the full power spectrum. The Spectrum Scope block only supports `One-sided ([0...Fs/2])` spectrums for real input signals.

Other DSP System Toolbox FFT-based blocks, including the blocks in the Power Spectrum Estimation library, always compute the FFT at frequencies in the range $[0,F_s)$.

Select the **Buffer input** check box to buffer the input signal. When the block buffers the input, the **Buffer size** parameter specifies the number of input samples to buffer before computing and displaying the magnitude FFT. You also use the **Buffer overlap** parameter to specify the number of samples from the previous buffer to include in the current buffer. To compute the number of new input samples the block acquires before computing and displaying the magnitude FFT, subtract the buffer overlap from the buffer size.

The display update period is

Spectrum Scope

$$(M_o - L) * T_s$$

where

- M_o = buffer size
- L = buffer overlap
- T_s = input sample period

For negative buffer overlap values, the block discards the appropriate number of input samples after the buffer fills. The block also updates the scope display at a slower rate than in the zero-overlap case.

Note If you are using the Spectrum Scope block inside an enabled subsystem and you select the **Buffer input** check box, you may get unexpected results. See "Behavior in Enabled Subsystems" on page 2-1394 for more information.

The **Window** and **Window sampling** parameters apply to the specification of the window function. See the Window Function block reference page for more details on these parameters.

The block determines the FFT length, N_{fft}, in the following ways:

- If you clear the **Specify FFT length** check box and select **Buffer input**, the block uses the buffer size as the FFT size.
- If you clear both the **Specify FFT length** and **Buffer input** check boxes, the block uses the input size as the FFT size.
- If you select the **Specify FFT length** check box, the **FFT length** parameter appears in the dialog box. Enter the number of samples on which you want the block to perform the FFT. This value must be a power of two.

The block zero pads or wraps the buffer of each channel to the FFT length before computing the FFT.

The value of the **Number of spectral averages** parameter determines the number of spectra to average. Setting this parameter to 1 effectively disables averaging. See the Periodogram block reference page for more information.

Display Properties Pane

The **Display Properties** pane enables you to control how the block displays your data.

The **Show grid** parameter toggles the background grid on and off.

If you select the **Persistence** check box, the window maintains successive displays. That is, the scope does not erase the display after each frame (or collection of frames), but overlays successive input frames in the scope display.

If you select the **Frame number** check box, the block displays the number of the current frame in the input sequence on the scope window, and the block increments the count as each new input is received. Counting starts at 1 with the first input frame, and continues until the simulation stops.

If you select the **Channel legend** check box, a legend indicating the line color, style, and marker of each channel's data is added. When the input signal is labeled, that label appears in the channel legend. When the input signal is not labeled, but comes from a Concatenate block or a Mux block with labeled inputs, those labels appear in the channel legend. Otherwise, each channel in the legend is labeled with the channel number (CH 1, CH 2, etc.). Click and drag the legend to reposition it in the scope window; double-click the line label to edit the text. If you rerun the simulation, the labels revert to the defaults.

If you select the **Compact display** check box, the scope completely fills the figure window. The scope does not display menus and axis titles, but it does show the numerical axis labels within the axes. If you clear the **Compact display** check box, the scope displays the axis labels and

Spectrum Scope

titles in a gray border surrounding the scope axes, and the window's menus and toolbar become visible.

If you select the **Open scope at start of simulation** check box, the scope opens at the start of the simulation. If you clear this parameter, the scope does not open automatically during the simulation. You can use this feature when you have several scope blocks in a model, and you do not want to view all the associated scopes during the simulation.

To view a scope window that is not open during simulation, click **Open scope immediately** on the **Display Properties** pane of the desired Scope block.

Note Before running a model that contains a Spectrum Scope block in Accelerator, Rapid Accelerator, or External mode, you must select the **Open scope at start of simulation** check box. If you do not select this check box before running your model for the first time, the scope will not display your simulation data.

The **Scope position** parameter specifies a four-element vector of the form

```
[left bottom width height]
```

specifying the position of the scope window on the screen, where (0,0) is the lower-left corner of the display. See the MATLAB `figure` function for more information.

Axis Properties Pane

If you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input to the block. For the block to produce valid output, the following conditions must 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 equals the sample period of the original time series.

In cases where not all these conditions hold, specify the appropriate value for the **Sample time of original time series** parameter.

When you set the **Frequency display limits** to Auto, the block displays the full spectrum over the frequency range specified by the **Spectrum type** parameter.

The **Frequency display offset** parameter allows you to offset the range of values displayed on the frequency axis of the Spectrum Scope.

- When the **Frequency display offset** is 0, the block displays the DC frequency (0 Hz) at 0 Hz.

- When **Frequency display offset** is a nonzero value, the block displays the DC frequency (0 Hz) at the value specified in the **Frequency display offset** parameter. If you set the **Frequency display limits** parameter to User-defined, the block does not automatically relabel the frequency axis. However, if you set the **Frequency display limits** to Auto, the values displayed on the frequency axis shift according to the **Frequency display offset** parameter.

  For example, if the block has the following settings:

  - **Spectrum Units** = Watts/Hertz
  - **Spectrum Type** = Two-sided ((-Fs/2...Fs/2])
  - **Frequency display offset (Hz)** = 0
  - **Frequency display limits** = Auto
  - Sampling frequency (*Fs*) = 1000 Hz

  Then, based on these settings:

  - The values on the frequency axis of the spectrum scope range from -500 Hz to 500 Hz.
  - The block centers the DC frequency (0 Hz) at 0 Hz.

Spectrum Scope

If you change the **Frequency display offset (Hz)** parameter to 100, the block:

- Relabels the frequency axis such that the values range from -400 Hz to 600 Hz.

- Centers the DC frequency (0 Hz) at 100 Hz.

When you set the **Frequency display limits** to User-defined, the **Minimum frequency (Hz)** and **Maximum frequency (Hz)** parameters set the range of the horizontal axis.

Minimum Y-limit and **Maximum Y-limit** parameters allow you to set the range of the vertical axis. Setting these parameters equates to setting the ymin and ymax values of the MATLAB axis function.

The **Y-axis label** is the text displayed to the left of the y-axis.

Line Properties Pane

Use the parameters on the **Line Properties** pane to help you distinguish between two or more independent channels of data on the scope.

The **Line visibilities** parameter specifies which channel's data is displayed on the scope, and which is hidden. The syntax specifies the visibilities in list form, where the term on or off as a list entry specifies the visibility of the corresponding channel's data. The list entries are separated by the pipe symbol, |.

For example, a five-channel signal would ordinarily generate five distinct plots on the scope. To disable plotting of the third and fifth lines, enter the following visibility specification in the **Line visibilities** parameter.

```
 on | on | off | on | off
ch 1  ch 2  ch 3  ch 4   ch 5
```

Note that the first (leftmost) list item corresponds to the first signal channel (leftmost column of the input matrix).

The **Line styles** parameter specifies the line style with which each channel's data is displayed on the scope. The syntax specifies the channel line styles in list form, with each list entry specifying a style for the corresponding channel's data. The list entries are separated by the pipe symbol, |.

For example, a five-channel signal would ordinarily generate all five plots with a solid line style. To plot each line with a different style, enter

```
  - | -- | : | -. | -
ch 1  ch 2  ch 3 ch 4  ch 5
```

These settings plot the signal channels with the following styles.

Line Style	Command to Type in Line Style Parameter	Appearance
Solid	-	
Dashed	--	
Dotted	:	
Dash-dot	-.	
No line	none	No line appears

Note that the first (leftmost) list item, '-', corresponds to the first signal channel (leftmost column of the input matrix). See the LineSpec property of the MATLAB line function for more information about the available markers.

The **Line markers** parameter specifies the marker style with which each channel's samples are represented on the scope. The syntax specifies the channels' marker styles in list form, with each list entry

specifying a marker for the corresponding channel's data. The list entries are separated by the pipe symbol, |.

For example, a five-channel signal would ordinarily generate all five plots with no marker symbol (that is, the individual sample points are not marked on the scope). To instead plot each line with a different marker style, you could enter

```
  *  |  .  |  x  |  s  |  d
ch 1    ch 2   ch 3   ch 4   ch 5
```

These settings plot the signal channels with the following styles.

Marker Style	Command to Type in Marker Style Parameter	Appearance
Asterisk	*	
Point	.	
Cross	x	
Square	s	
Diamond	d	

Note that the leftmost list item, '*', corresponds to the first signal channel or leftmost column of the input matrix. See the LineSpec property of the MATLAB line function for more information about the available markers.

To produce a stem plot for the data in a particular channel, type the word stem instead of one of the basic marker shapes.

The **Line colors** parameter specifies the color in which each channel's data is displayed on the scope. The syntax specifies the channel colors in list form, with each list entry specifying a color (in one of the MATLAB ColorSpec formats) for the corresponding channel's data. The list entries are separated by the pipe symbol, |.

For example, a five-channel signal would ordinarily generate all five plots in the color black. To instead plot the lines with the color order below, enter

```
[0 0 0] | [0 0 1] | [1 0 0 ] | [0 1 0] | [.7529 0 .7529]
  ch 1       ch 2        ch 3        ch 4           ch 5
```

or

```
'k' | 'b' | 'r' | 'g' | [.7529 0 .7529]
ch 1   ch 2   ch 3   ch 4          ch 5
```

These settings plot the signal channels in the following colors (8-bit RGB equivalents shown in the center column).

Color	RGB Equivalent	Appearance
Black	(0,0,0)	
Blue	(0,0,255)	
Red	(255,0,0)	
Green	(0,255,0)	
Dark purple	(192,0,192)	

Spectrum Scope

Note that the leftmost list item, `'k'`, corresponds to the first signal channel or leftmost column of the input matrix. See the MATLAB function `ColorSpec` for more information about the color syntax.

Spectrum Scope Window

The title that appears in the title bar of the scope window is the same as the block title. In addition to the standard MATLAB figure window menus such as **File**, **Window**, and **Help**, the Spectrum Scope window contains **View**, **Axes**, and **Channels** menus.

The options in the **View** menu allow you to zoom in and out of the scope window:

- To zoom in on the scope window, you must first select **View > Zoom In** or click the corresponding `Zoom In` toolbar button (⊕). You can then zoom in by clicking in the center of your area of interest, or by clicking and dragging your cursor to draw a rectangular area of interest inside of the scope window.

- To zoom in on the *x*-axis of the scope window, you must first select **View > Zoom X** or click the corresponding `Zoom X-Axis` toolbar button (⊗) on the scope window. You can then zoom in on the *x*-axis with a single click inside the scope window, or by clicking and dragging the cursor along the *x*-axis over your area of interest.

- To zoom in on the *y*-axis of the scope window, you must first select **View > Zoom Y** or click the corresponding `Zoom Y-Axis` toolbar button (⊘). You can then zoom in on the *y*-axis with a single click inside the scope window, or by clicking and dragging the cursor along the *y*-axis over your area of interest.

- To return to the original view of the scope window, you have the following options:

  - Select **Full View** from the **View** menu.

  - Click the `Restore default view` toolbar button (⤢) on the Spectrum Scope window.

- Right-click inside the scope window and select `Reset to Original View`.

Note To zoom out in smaller increments, you can right-click inside of the scope window and select `Zoom Out`. You can also zoom out by holding down the **Shift** key and clicking the left mouse button inside the scope window.

The parameters that you set using the **Axes** menu apply to all channels. Many of the parameters in this menu are also accessible through the block parameters dialog box. For descriptions of these parameters, see "Display Properties Pane" on page 2-1399. Below are descriptions of other parameters in the **Axes** menu:

- **Refresh** erases all data on the scope display, except for the most recent trace. This command is useful in conjunction with the **Persistence** setting.

- **Autoscale** resizes the *y*-axis to best fit the vertical range of the data.

Note The **Minimum Y-limit** and **Maximum Y-limit** parameters on the **Axis properties** pane of the block dialog are not updated to display the numerical limits selected by the autoscale feature.

- **Save Axes Settings** allows you to save the current axes settings. When you select this option, the **Minimum Y-limit** and **Maximum Y-limit** parameters of the **Axes Properties** pane update with the current *y*-axes limits. The **Frequency display limits** parameter is set to `User-defined`, and the current *x*-axes limits are saved in the **Minimum Frequency (Hz)** and **Maximum Frequency (Hz)** parameters. To save these axes settings for your next MATLAB session, you need to resave your model.

- **Save Scope Position** automatically updates the **Scope position** parameter in the **Axis Properties** pane of the block dialog. When you select **Save Scope Position**, the block saves the current position and size of the scope window. To make the scope window open at a particular location on the screen when the simulation runs, drag the window to the desired location, resize it, and select **Save Scope Position** from the **Axes** menu.

The properties listed in the **Channels** menu apply to a particular channel. All of the parameters in this menu are also accessible through the block parameters dialog box. For descriptions of these parameters, see "Line Properties Pane" on page 2-1388.

Many of these options are also accessible by right-clicking the mouse anywhere on the scope display. The menu that is displayed contains a combination of the options available in the **View**, **Axes** and **Channels** menus.

Note When you select **Compact Display** from the **Axes** menu, the scope window menus are no longer visible. Right-click in the Spectrum Scope window and click **Compact Display** to make the menus reappear.

Behavior in Enabled Subsystems

When you clear the **Buffer input** check box, the Spectrum Scope block is fully supported inside of "Enabled Subsystems". However, when you choose to buffer the input to the Spectrum Scope block, you may get unexpected results inside of enabled subsystems. For more information about the limitations of buffering signals inside of enabled subsystems, see the "Behavior in Enabled Subsystems" on page 2-172 section of the Buffer block reference page.

Examples   See "Display Frequency-Domain Data" in the *DSP System Toolbox User's Guide*.

Dialog Box

Scope Properties Pane
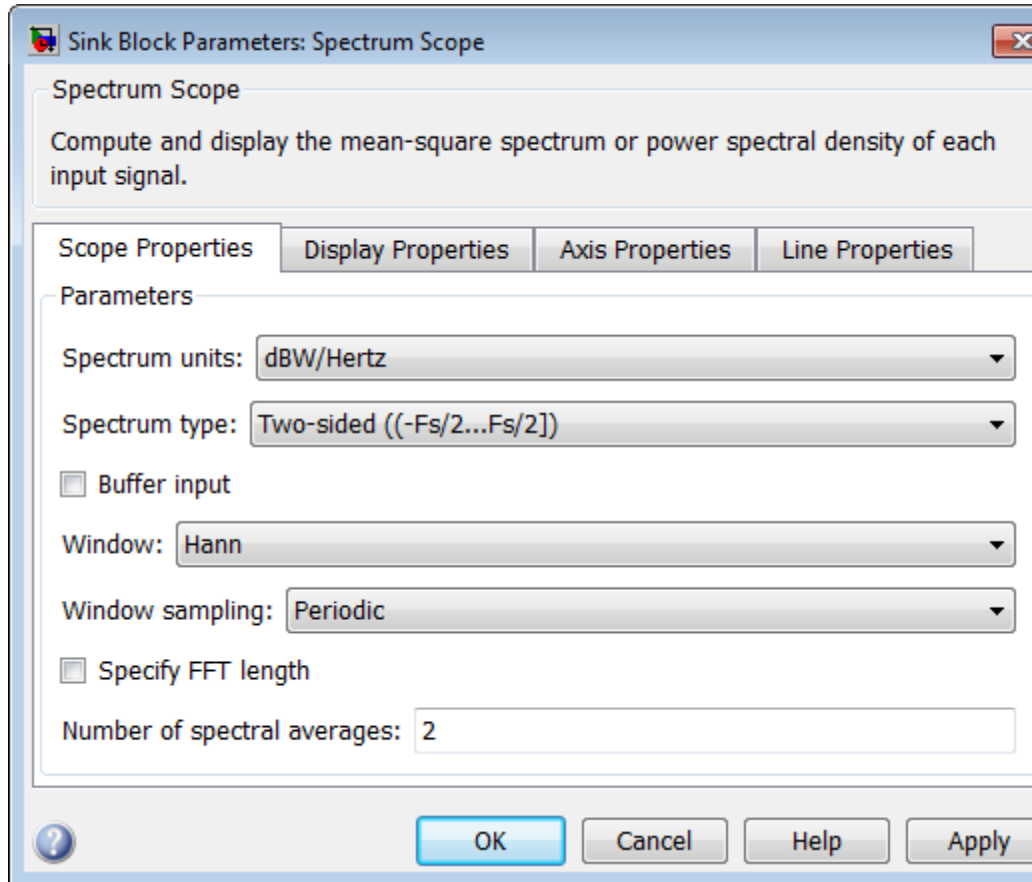
Spectrum units

Specify the spectrum units as described in the following table. The specified units determine the type of measurement to compute (Mean-Square Spectrum or Power Spectral Density). They also determine the type of Y-axis scaling (linear or logarithmic).

Spectrum Scope

Spectrum Units	Measurement Type	Scaling
Watts	Mean-Square Spectrum (MSS)	Linear
dBW dBm	Mean-Square Spectrum (MSS)	Logarithmic
Watts/Hertz	Power Spectral Density (PSD)	Linear
dBW/Hertz dBm/Hertz	Power Spectral Density (PSD)	Logarithmic

You can only tune this parameter within the same **Measurement type**. The block cannot switch between computing the mean-square spectrum and the power spectral density while the simulation is running.

Spectrum type
Specify the range of frequencies over which to compute the magnitudes in the input. The available options are One-sided ([0...Fs/2]) and Two-sided ((-Fs/2...Fs/2)), where F_s is the sampling frequency of the original frequency-domain signal. If you select One-sided ([0...Fs/2]), the input signal must be real-valued. Tunable.

Buffer input
Select this check box to rebuffer the input data.

If you are using the Spectrum Scope block inside of an enabled subsystem, you may get unexpected results. See "Behavior in Enabled Subsystems" on page 2-1394 for more information.

The Spectrum Scope block does not support buffering in models running in external mode. To buffer the input when running your

model in external mode, clear this check box and place a Buffer block before the Spectrum Scope block in your model.

Buffer size

Specify the number of input samples that the block buffers before computing and displaying the magnitude FFT. You must specify a real integer value greater than or equal to two. If you do not select the **Specify FFT length** check box, the **Buffer size** must be a power of two.

This parameter becomes visible only when you select the **Buffer input** check box.

Buffer overlap

Specify the number of samples from the previous buffer to include in the current buffer. To compute the number of new input samples the block acquires before computing and displaying the magnitude FFT, subtract the buffer overlap from the buffer size.

This parameter becomes visible only when you select the **Buffer input** check box.

Treat Mx1 and unoriented sample-based signals as

Specify how the block treats sample-based M-by-1 column vectors and unoriented sample-based vectors of length M. You can select one of the following options:

- `One channel` — When you select this option, the block treats M-by-1 and unoriented sample-based inputs as a column vector (one channel).

- `M channels` — When you select this option, the block treats M-by-1 and unoriented sample-based inputs as a 1-by-M row vector.

Window

Specify the type of window to apply. See the Window Function block reference page for more details. Tunable.

Stopband attenuation in dB

Enter the level, in decibels (dB), of stopband attenuation, R_s, for the Chebyshev window. Tunable.

This parameter becomes visible only when you select `Chebyshev` for the **Window** parameter.

Beta

Enter the β parameter for the Kaiser window. Increasing **Beta** widens the mainlobe and decreases the amplitude of the sidelobes in the displayed frequency magnitude response. Tunable.

This parameter becomes visible only if you select `Kaiser` for the **Window** parameter.

Window sampling

Choose `Symmetric` or `Periodic`. Tunable.

This parameter becomes visible only if `Blackman`, `Hamming`, `Hann`, or `Hanning` is selected for the **Window** parameter.

Specify FFT length

Select this check box to specify the FFT length yourself in the **FFT length** parameter.

FFT length

Enter the number of samples on which you want the block to perform the FFT. The value you specify must be a power of two.

This parameter becomes visible only when you select the **Specify FFT length** check box.

Number of spectral averages

The number of spectra to average. Setting this parameter to 1 effectively disables averaging. See the Periodogram block reference page for more information.

Display Properties Pane

Show grid

Toggle the scope grid on and off. Tunable.

Persistence

Select this check box to maintain successive displays. That is, the scope does not erase the display after each frame (or collection

of frames), but overlays successive input frames in the scope display. Tunable.

Frame number

If you select this check box, the number of the current frame in the input sequence appears in the Vector Scope window. Tunable.

Channel legend

Toggles the legend on and off. Tunable.

Compact display

Resizes the scope to fill the window. Tunable.

Open scope at start of simulation

Select this check box to open the scope at the start of the simulation. When you clear this parameter, the scope does not automatically open during the simulation. Tunable.

Note Before running a model that contains a Spectrum Scope block in Accelerator, Rapid Accelerator, or External mode, you must select the **Open scope at start of simulation** check box. If you do not select this check box before running your model for the first time, the scope will not display your simulation data.

Open scope immediately

If the scope is not open during simulation, select this check box to open it. This parameter is visible only while the simulation is running.

Scope position

A four-element vector of the form [left bottom width height] specifying the position of the scope window. (0,0) is the lower-left corner of the display. Tunable.

Axis Properties Pane

Inherit sample time from input

If you select this check box, the block computes the time-domain sample period from the frame period and frame size of the frequency-domain input. Use this parameter only when the length of each frame of frequency-domain data equals the frame length of the time-domain data that generated it.

Spectrum Scope

> **Note** The block cannot inherit sample time when you specify a non-zero value for the **Buffer overlap** parameter and set the **Spectrum units** parameter to `dBW/Hertz`, `Watts/Hertz` or `dBm/Hertz`. You must specify the sample time by clearing this check box and using the **Sample time of original time series** parameter.

Sample time of original time series
> Enter the sample period of the original time-domain signal.

Frequency display offset
> The **Frequency display offset** parameter allows you to offset the range of values displayed on the frequency axis of the Spectrum Scope. The value specified in this field becomes the new label for the DC frequency (0 Hz). See the example in the "Axis Properties Pane" on page 2-1386 section for more information.

Frequency display limits
> Select `Auto` to have the limits of the *x*-axis set for you automatically. Select `User-defined` to set the limits yourself in the **Minimum frequency** and **Maximum frequency** parameters.

Minimum frequency (Hz)
> Specify the minimum frequency value of the *x*-axis in Hertz. This parameter is only visible if the **Frequency display limits** parameter is set to `User-defined`. Tunable.

Maximum frequency (Hz)
> Specify the maximum frequency value of the *x*-axis in Hertz. This parameter is only visible if the **Frequency display limits** parameter is set to `User-defined`. Tunable.

Minimum Y-limit
> Specify the minimum value of the *y*-axis. Setting this parameter is analogous to setting the `ymin` value of the MATLAB `axis` function. Tunable.

Maximum Y-limit

Specify the maximum value of the *y*-axis. Setting this parameter is analogous to setting the `ymax` value of the MATLAB `axis` function. Tunable.

Y-axis label

Specify text for the block to display to the left of the *y*-axis. Tunable.

Spectrum Scope

Line Properties Pane
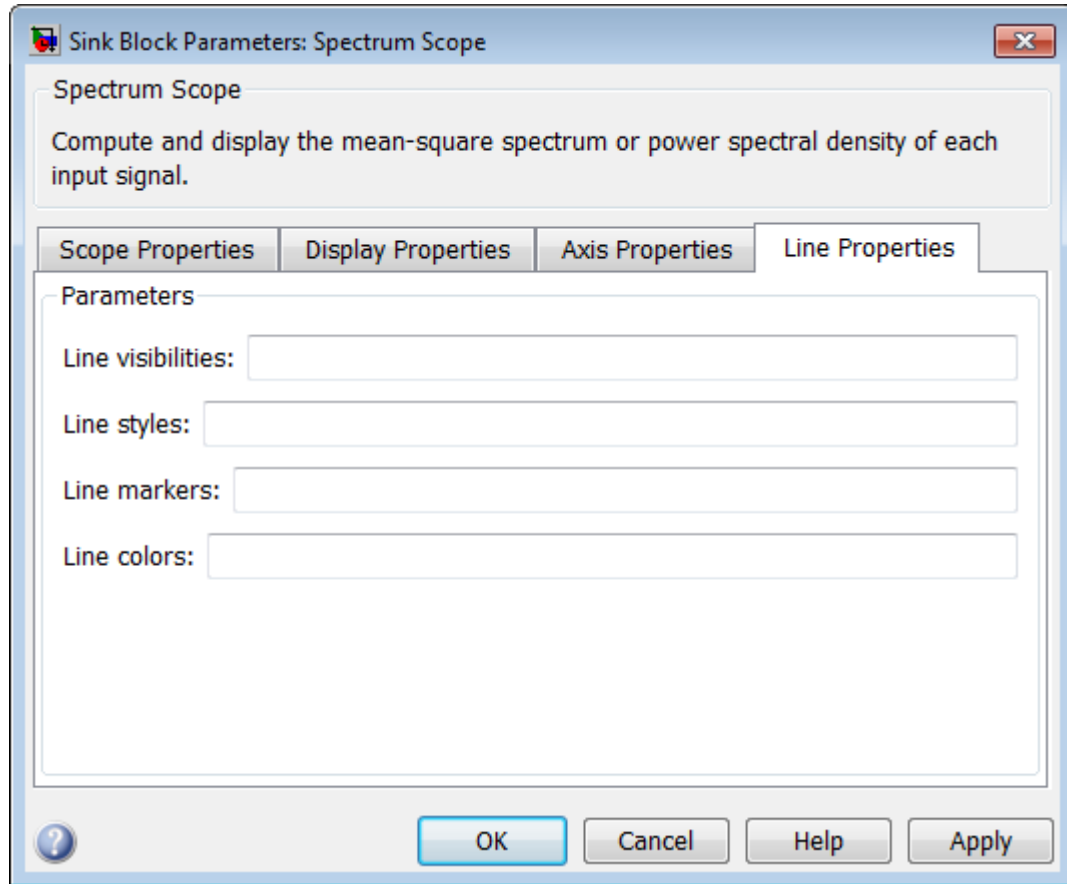
Line visibilities

Enter on or off to specify the visibility of the scope traces for various channels. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Line styles
Enter the line styles of the scope traces for various channels using the MATLAB `line` function `LineStyle` formats. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Line markers
Enter the line markers of the scope traces for various channels using the MATLAB `line` function `Marker` formats. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Line colors
Enter the colors of the scope traces for various channels using the MATLAB `ColorSpec` formats. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

FFT	DSP System Toolbox
Periodogram	DSP System Toolbox
Short-Time FFT	DSP System Toolbox
Vector Scope	DSP System Toolbox
Window Function	DSP System Toolbox

Stack

Purpose   Store inputs into LIFO register

Library   Signal Management / Buffers

dspbuff3

Description
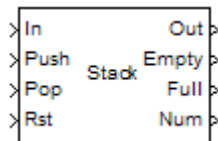
The Stack block stores a sequence of input samples in a last in, first out (LIFO) register. The register capacity is set by the **Stack depth** parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the In port onto the top of the stack when a trigger event is received at the Push port. When a trigger event is received at the Pop port, the block *pops* the top element off the stack and holds the Out port at that value. The last input to be pushed onto the stack is always the first to be popped off.
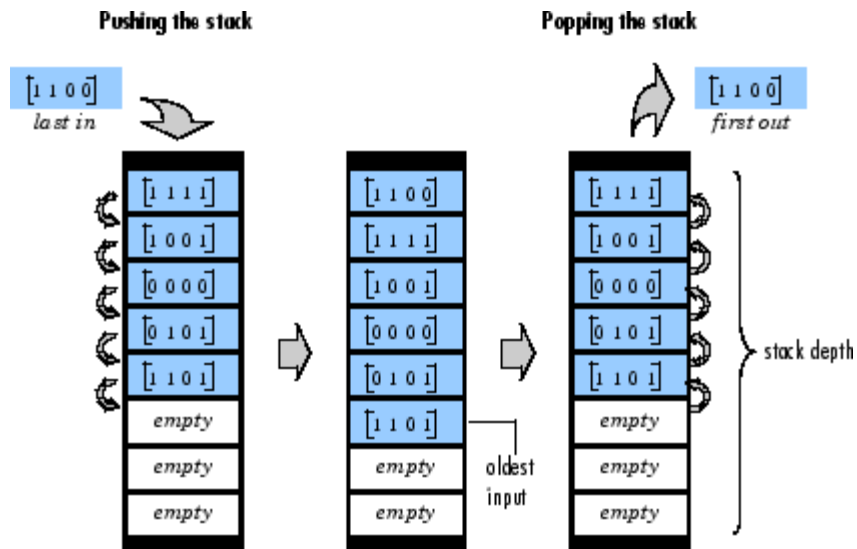
A trigger event at the optional Rst port empties the stack contents. When you select **Clear output port on reset**, then a trigger event at the Rst port empties the stack *and* sets the value at the Out port to zero. This setting also applies when a disabled subsystem containing

the Stack block is reenabled; the Out port value is only reset to zero in this case when you select **Clear output port on reset**.

When two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

1 Rst

2 Push

3 Pop

The rate of the trigger signal must be the same as the rate of the data signal input. You specify the triggering event for the Push, Pop, and Rst ports in the **Trigger type** pop-up menu:

- Rising edge — Triggers execution of the block when the trigger input does one of the following:

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

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

- Falling edge — Triggers execution of the block when the trigger input does one of the following:

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

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

Not a falling edge because it continues a fall from a positive value to zero.

- `Either edge` — Triggers execution of the block when the trigger input is a `Rising edge` or `Falling edge` (as described above).

- `Non-zero sample` — Triggers execution of the block at each sample time that the trigger input is not zero.

Note If your model contains any referenced models that use a Stack block with the **Push full stack** parameter set to `Dynamic reallocation`, you cannot simulate your top-level model in Simulink Accelerator mode.

The **Push full stack** parameter specifies the block's behavior when a trigger is received at the Push port but the register is full. The **Pop empty stack** parameter specifies the block's behavior when a trigger is received at the Pop port but the register is empty. The following options are available for both cases:

- `Ignore` — Ignore the trigger event, and continue the simulation.

- Warning — Ignore the trigger event, but display a warning message in the MATLAB command window.

- Error — Display an error dialog box and terminate the simulation.

> **Note** The **Push full stack** and **Pop empty stack** parameters are diagnostic parameters. Like all diagnostic parameters on the Configuration Parameters dialog box, they are set to Ignore in the code generated for this block by Simulink Coder code generation software.

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 **Show number of stack entries port** parameter.

> **Note** When Dynamic reallocation is selected, the **System target file** parameter on the **Code Generation** pane of the Model Configuration Parameters dialog box must be set to grt_malloc.tlc Generic Real-Time Target with dynamic memory allocation.
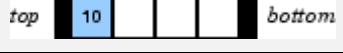
Examples

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 Rst 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	Rst	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 Rst ports are triggered simultaneously. The Rst trigger takes precedence, and the stack is first cleared and then pushed.

Example 2

The dspqdemo example provides an example of the related Queue block.

Dialog Box
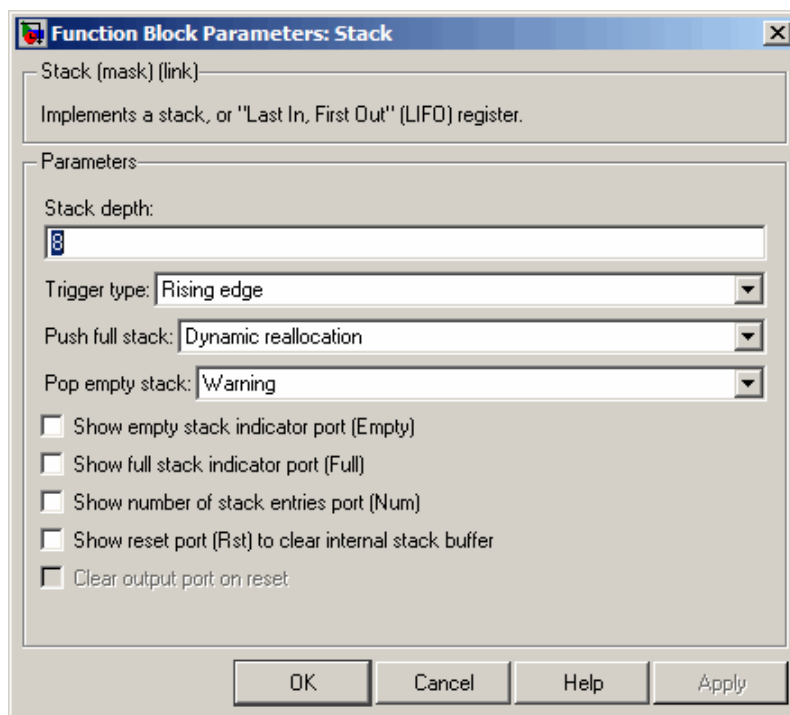
Stack depth
The number of entries that the LIFO register can hold.

Trigger type
The type of event that triggers the block's execution. The rate of the trigger signal must be the same as the rate of the data signal input.

Push full stack
Response to a trigger received at the Push port when the register is full. Inputs to this port must have the same built-in data type as inputs to the Pop and Rst input ports.

When `Dynamic reallocation` is selected, the **System target file** parameter on the **Code Generation** pane of the Model Configuration Parameters dialog box must be set to `grt_malloc.tlc Generic Real-Time Target with dynamic memory allocation`.

Pop empty stack

Response to a trigger received at the `Pop` port when the register is empty. Inputs to this port must have the same built-in data type as inputs to the `Push` and `Rst` input ports.

Show empty stack indicator port

Enable the `Empty` output port, which is high (`1`) when the stack is empty, and low (`0`) otherwise.

Show full stack indicator port

Enable the `Full` output port, which is high (`1`) when the stack is full, and low (`0`) otherwise. The `Full` port remains low when you select **Dynamic reallocation** from the **Push full stack** parameter.

Show number of stack entries port

Enable the `Num` output port, which tracks the number of entries currently on the stack. When inputs to the `In` port are double-precision values, the outputs from the `Num` port are double-precision values. Otherwise, the outputs from the `Num` port are 32-bit unsigned integer values.

Show reset port to clear internal stack buffer

Enable the `Rst` input port, which empties the stack when the trigger specified by the **Trigger type** is received. Inputs to this port must have the same built-in data type as inputs to the `Push` and `Pop` input ports.

Clear output port on reset

Reset the `Out` port to zero (in addition to clearing the stack) when a trigger is received at the `Rst` input port.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Push	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Pop and Rst input ports
Pop	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Push and Rst input ports.

Stack

Port	Supported Data Types
Rst	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Push and Pop input ports.
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Empty	• Double-precision floating point • Boolean
Full	• Double-precision floating point • Boolean
Num	• Double-precision floating point The block outputs a double-precision floating-point value at this port when the data type of the In port is double-precision floating-point. • 32-bit unsigned integers The block outputs a 32-bit unsigned integer value at this port when the data type of the In port is anything other than double-precision floating-point.

See Also

Buffer	DSP System Toolbox
Delay Line	DSP System Toolbox
Queue	DSP System Toolbox

Standard Deviation

Purpose   Find standard deviation of input or sequence of inputs

Library   Statistics

dspstat3
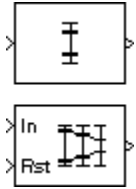
Description   The Standard Deviation block computes the standard deviation of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input. The Standard Deviation block can also track the standard deviation of a sequence of inputs over a period of time. The **Running standard deviation** parameter selects between basic operation and running operation.

Basic Operation

When you do not select the **Running standard deviation** check box, the block computes the standard deviation of each row or column of the input, along vectors of a specified dimension of the input, or of the entire input at each individual sample time, and outputs the array *y*. Each element in *y* contains the standard deviation of the corresponding column, row, vector, or entire input. The output *y* depends on the setting of the **Find the standard deviation value over** parameter. For example, consider a 3-dimensional input signal of size M-by-N-by-P:

- Entire input — The output at each sample time is a scalar that contains the standard deviation of the entire input.

  ```
  y = std(u(:))     % Equivalent MATLAB code
  ```

- Each Row — The output at each sample time consists of an M-by-1-by-P array, where each element contains the standard deviation of each vector over the second dimension of the input. For an input that is an M-by-N matrix, the output at each sample time is an M-by-1 column vector.

  ```
  y = std(u,0,2)     % Equivalent MATLAB code
  ```

- `Each Column` — The output at each sample time consists of a 1-by-N-by-P array, where each element contains the standard deviation of each vector over the first dimension of the input. For an input that is an M-by-N matrix, the output at each sample time is a 1-by-N row vector.

  ```
  y = std(u,0,1)     % Equivalent MATLAB code
  ```

  In this mode, the block treats length-M unoriented vector inputs as M-by-1 column vectors.

- `Specified Dimension` — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as when you select `Each column`. If **Dimension** is set to 2, the output is the same as when you select `Each row`. If **Dimension** is set to 3, the output at each sample time is an M-by-N matrix containing the standard deviation of each vector over the third dimension of the input.

  ```
  y = std(u,0,Dimension)      % Equivalent MATLAB code
  ```

For purely real or purely imaginary inputs, the standard deviation of the jth column of an M-by-N input matrix is the square root of its variance:

$$y_j = \sigma_j = \sqrt{\frac{\sum_{i=1}^{M} \left| u_{ij} - \mu_j \right|^2}{M-1}} \qquad 1 \le j \le N$$

For complex inputs, the output is the *total standard deviation*, which equals the square root of the *total variance*, or the square root of the sum of the variances of the real and imaginary parts. The standard deviation of each column in an M-by-N input matrix is given by:

$$\sigma_j = \sqrt{\sigma_{j,\text{Re}}^2 + \sigma_{j,\text{Im}}^2}$$

Standard Deviation

> **Note** The total standard deviation does *not* equal the sum of the real and imaginary standard deviations.

Running Operation

When you select the **Running standard deviation** check box, the block tracks the standard deviation of successive inputs to the block. In this mode, you must also specify a value for the **Input processing** parameter:

- When you select Elements as channels (sample based), the block outputs an *M*-by-*N* array. Each element y_{ij} of the output contains the standard deviation of the element u_{ij} over all inputs since the last reset.

- When you select Columns as channels (frame based), the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the standard deviation of the *j*th column over all inputs since the last reset, up to and including element u_{ij} of the current input.

Running Operation for Variable-Size Inputs

When your inputs are of variable size, and you select the **Running standard deviation** check box, there are two options:

- If you set the **Input processing** parameter to Elements as channels (sample based), the state is reset.

- If you set the **Input processing** parameter to Columns as channels (frame based), then there are two cases:

  - When the input size difference is in the number of channels (i.e., number of columns), the state is reset.

  - When the input size difference is in the length of channels (i.e., number of rows), there is no reset and the running operation is carried out as usual.

Resetting the Running Standard Deviation

The block resets the running standard deviation whenever a reset event is detected at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

You specify the reset event in the **Reset port** parameter:

- None disables the Rst port.

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

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

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

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

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

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

Standard Deviation

Falling edge   Falling edge

Falling edge

Falling edge

Not a falling edge because it continues a fall from a positive value to zero.

- Either edge — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described earlier)

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

Note When running simulations in the Simulink MultiTasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

ROI Processing

To calculate the statistical value within a particular region of interest (ROI) of the input, select the **Enable ROI processing** check box. This option is only available when the **Find the standard deviation value over** parameter is set to Entire input and the **Running standard deviation** check box is not selected. ROI processing is only supported for 2-D inputs.

Note Full ROI processing is available only if you have a Computer Vision System Toolbox license. If you do not have a Computer Vision System Toolbox license, you can still use ROI processing, but are limited to the **ROI type** Rectangles.

Use the **ROI type** parameter to specify whether the ROI is a rectangle, line, label matrix, or binary mask. A binary mask is a binary image that enables you to specify which pixels to highlight, or select. In a label matrix, pixels equal to 0 represent the background, pixels equal to 1 represent the first object, pixels equal to 2 represent the second object, and so on. When the **ROI type** parameter is set to Label matrix, the Label and Label Numbers ports appear on the block. Use the Label Numbers port to specify the objects in the label matrix for which the block calculates statistics. The input to this port must be a vector of scalar values that correspond to the labeled regions in the label matrix. For more information about the format of the input to the ROI port when the ROI is a rectangle or a line, see the Draw Shapes block reference page.

For rectangular ROIs, use the **ROI portion to process** parameter to specify whether to calculate the statistical value for the entire ROI or just the ROI perimeter.

Use the **Output** parameter to specify the block output. The block can output separate statistical values for each ROI or the statistical value for all specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select Binary mask.

If, for the **ROI type** parameter, you select Rectangles or Lines, the **Output flag indicating if ROI is within image bounds** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

Standard Deviation

Output = Individual statistics for each ROI

Flag Port Output	Description
0	ROI is completely outside the input image.
1	ROI is completely or partially inside the input image.

Output = Single statistic for all ROIs

Flag Port Output	Description
0	All ROIs are completely outside the input image.
1	At least one ROI is completely or partially inside the input image.

If the ROI is partially outside the image, the block only computes the statistical values for the portion of the ROI that is within the image.

If, for the **ROI type** parameter, you select Label matrix, the **Output flag indicating if input label numbers are valid** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

Output = Individual statistics for each ROI

Flag Port Output	Description
0	Label number is not in the label matrix.
1	Label number is in the label matrix.

Output = Single statistic for all ROIs

Flag Port Output	Description
0	None of the label numbers are in the label matrix.
1	At least one of the label numbers is in the label matrix.

Examples

In the following ex_standarddeviation_ref model, the Standard Deviation block calculates the running standard deviation of a 3-by-2 matrix input, u. The **Input processing** parameter is set to Columns as channels (frame based), so the block processes the input as a two channel signal with a frame size of three. The running standard deviation is reset at *t*=2 by an impulse to the block's Rst port.
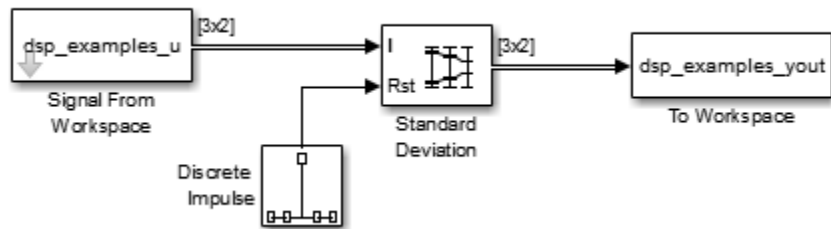
The operation of the block is shown in the following figure.

Standard Deviation
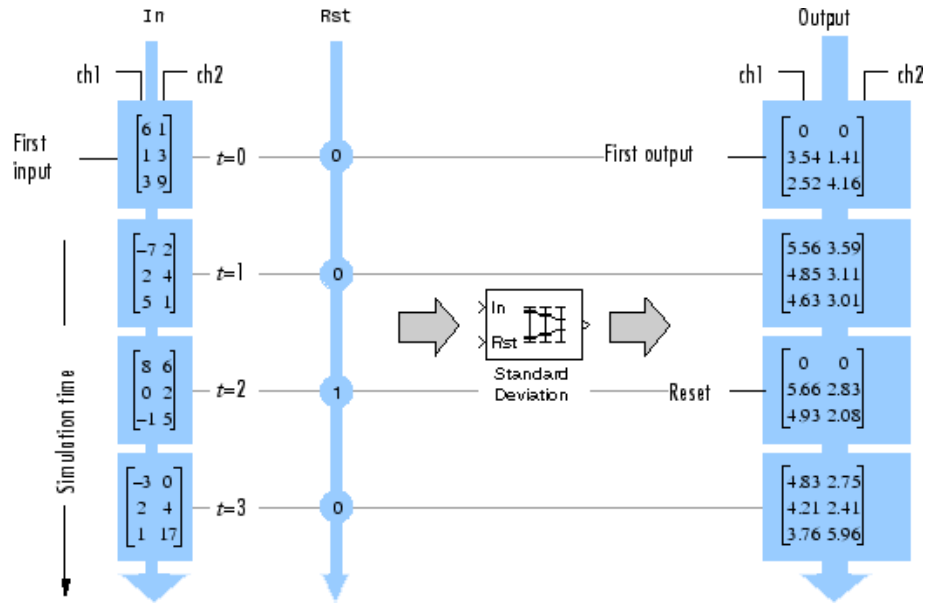
Dialog Box
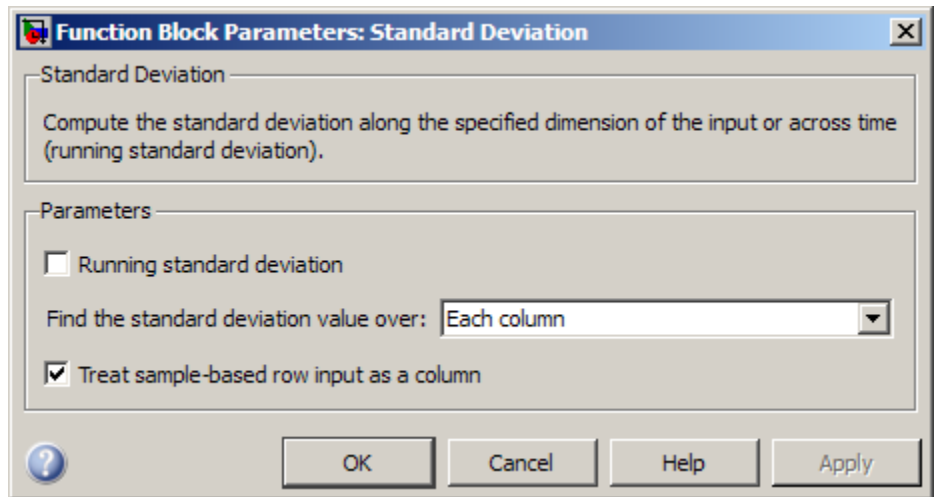
Running standard deviation
Enables running operation when selected.

Input processing
Specify how the block should process the input when computing the running standard deviation. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

This parameter appears only when you select the **Running standard deviation** check box.

Standard Deviation

> **Note** The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Reset port
> Specify the reset event that causes the block to reset the running standard deviation. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you select the **Running standard deviation** check box. For more information, see "Resetting the Running Standard Deviation" on page 2-1419.

Find the standard deviation value over
> Specify whether to find the standard deviation value along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see "Basic Operation" on page 2-1416.

Treat sample-based row input as a column
> Select to treat sample-based length-*M* row vector inputs as *M*-by-1 column vectors. This parameter is only visible when the **Find the standard deviation value over** parameter is set to `Each column`.

> **Note** This check box will be removed in a future release. See "Sample-Based Row Vector Processing Changes" in the DSP System Toolbox Release Notes.

Dimension
> Specify the dimension (one-based value) of the input signal, over which the standard deviation is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the**

standard deviation value over parameter is set to `Specified dimension`.

Enable ROI Processing

Select this check box to calculate the statistical value within a particular region of each image. This parameter is only available when the **Find the standard deviation value over** parameter is set to `Entire input`, and the block is not in running mode.

Note Full ROI processing is available only when you have a Computer Vision System Toolbox license. If you do not have a Computer Vision System Toolbox license, you can still use ROI processing, but are limited to the **ROI type** `Rectangles`.

ROI type

Specify the type of ROI you want to use. Your choices are `Rectangles`, `Lines`, `Label matrix`, or `Binary mask`.

ROI portion to process

Specify whether you want to calculate the statistical value for the entire ROI or just the ROI perimeter. This parameter is only visible if, for the **ROI type** parameter, you specify `Rectangles`.

Output

Specify the block output. The block can output a vector of separate statistical values for each ROI or a scalar value that represents the statistical value for all the specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select `Binary mask`.

Output flag

☑ **Output flag indicating if ROI is within image bounds**

☑ **Output flag indicating if label numbers are valid**

Standard Deviation

When you select either of these check boxes, the Flag port appears on the block. For a description of the Flag port output, see the tables in "ROI Processing" on page 2-1420.

The **Output flag indicating if ROI is within image bounds** check box is only visible when you select Rectangles or Lines as the **ROI type**.

The **Output flag indicating if label numbers are valid** check box is only visible when you select Label matrix for the **ROI type** parameter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Reset	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
ROI	Rectangles and lines: • Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Binary Mask:

Port	Supported Data Types
	• Boolean
Label	• 8-, 16-, and 32-bit unsigned integers
Label Numbers	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point
Flag	• Boolean

See Also

Mean	DSP System Toolbox
RMS	DSP System Toolbox
Variance	DSP System Toolbox
std	MATLAB

Submatrix

Purpose

Select subset of elements (submatrix) from matrix input

Library

- Math Functions / Matrices and Linear Algebra / Matrix Operations

  dspmtrx3

- Signal Management / Indexing

  dspindex

Description
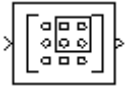
The Submatrix block extracts a contiguous submatrix from the *M*-by-*N* input matrix u. The block treats length-*M* unoriented vector input as an *M*-by-1 matrix. The **Row span** parameter provides three options for specifying the range of rows in u to be retained in submatrix output y:

- All rows

  Specifies that y contains all *M* rows of u.

- One row

  Specifies that y contains only one row from u. The **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 **Starting 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.

Range Specification Options

When you select One row or Range of rows from the **Row span** parameter, you specify the desired row or range of rows in the **Row** parameter, or the **Starting row** and **Ending row** parameters. Similarly, when you select One column or Range of columns from the

Column span parameter, you specify the desired column or range of columns in the **Column** parameter, or the **Starting column** and **Ending column** parameters.

The **Row**, **Column**, **Starting row** or **Starting column** can be specified in six ways:

- First

  For rows, this specifies that the first row of u should be used as the first row of y. When all columns are to be included, this is equivalent to `y(1,:)  = u(1,:)`.

  For columns, this specifies that the first column of u should be used as the first column of y. When all rows are to be included, this is equivalent to `y(:,1) = u(:,1)`.

- Index

  For rows, this specifies that the row of u, `firstrow`, forward-indexed by the **Row index** parameter or the **Starting row index** parameter, should be used as the first row of y. When all columns are to be included, this is equivalent to `y(1,:)  = u(firstrow,:)`.

  For columns, this specifies that the column of u, forward-indexed by the **Column index** parameter or the **Starting column index** parameter, `firstcol`, should be used as the first column of y. When all rows are to be included, this is equivalent to `y(:,1) = u(:,firstcol)`.

- Offset from last

  For rows, this specifies that the row of u offset from row *M* by the **Row offset** or **Starting row offset** parameter, `firstrow`, should be used as the first row of y. When all columns are to be included, this is equivalent to `y(1,:)  = u(M-firstrow,:)`.

  For columns, this specifies that the column of u offset from column *N* by the **Column offset** or **Starting column offset** parameter, `firstcol`, should be used as the first column of y. When all rows are to be included, this is equivalent to `y(:,1) = u(:,N-firstcol)`.

- Last

  For rows, this specifies that the last row of u should be used as the only row of y. When all columns are to be included, this is equivalent to y = u(M,:).

  For columns, this specifies that the last column of u should be used as the only column of y. When all rows are to be included, this is equivalent to y = u(:,N).

- Offset from middle

  When you select this option, the block selects the first row or column of the output y by adding the specified offset to the middle row or column of the input u. When the number, *X*, of input rows or columns is even, the block defines the middle one as *X*/2 +1. When the number of input rows or columns is odd, the block defines the middle one as ceil(*X*/2).

  When all columns are to be included, the following code defines the starting row: y(1,:) = u(MiddleRow+Offset,:), where Offset is the value of the **Row offset** or **Starting row offset** parameter. When all rows are to be included, the following code defines the starting column: y(1,:) = u(:,MiddleColumn+Offset), where Offset is the value of the **Column offset** or **Starting column offset** parameter.

- Middle

  When you select this option, the block uses the middle row or column of the input u as the first row or column of the output y. When the number, *X*, of input rows or columns is even, the block defines the middle one as *X*/2 +1. When the number of input rows or columns is odd, the block defines the middle one as ceil(*X*/2).

  When all columns are to be included, the following code defines the starting row: y = u(MiddleRow,:). When all rows are to be included, the following code defines the starting column: y = u(:,MiddleColumn).

The **Ending row** or **Ending column** can similarly be specified in five ways:

- Index

  For rows, this specifies that the row of u forward-indexed by the **Ending row index** parameter, `lastrow`, should be used as the last row of y. When all columns are to be included, this is equivalent to `y(end,:)  = u(lastrow,:)`.

  For columns, this specifies that the column of u forward-indexed by the **Ending column index** parameter, `lastcol`, should be used as the last column of y. When all rows are to be included, this is equivalent to `y(:,end) = u(:,lastcol)`.

- Offset from last

  For rows, this specifies that the row of u offset from row *M* by the **Ending row offset** parameter, `lastrow`, should be used as the last row of y. When all columns are to be included, this is equivalent to `y(end,:)  = u(M-lastrow,:)`.

  For columns, this specifies that the column of u offset from column *N* by the **Ending column offset** parameter, `lastcol`, should be used as the last column of y. When all rows are to be included, this is equivalent to `y(:,end) = u(:,N-lastcol)`.

- Last

  For rows, this specifies that the last row of u should be used as the last row of y. When all columns are to be included, this is equivalent to `y(end,:)  = u(M,:)`.

  For columns, this specifies that the last column of u should be used as the last column of y. When all rows are to be included, this is equivalent to `y(:,end) = u(:,N)`.

- Offset from middle

  When you select this option, the block selects the last row or column of the output y by adding the specified offset to the middle row or column of the input u. When the number, *X*, of input rows or columns

Submatrix

is even, the block defines the middle one as *X*/2 +1. When the number of input rows or columns is odd, the block defines the middle one as ceil(*X*/2).

When all columns are to be included, the following code defines the ending row: y(end,:) = u(MiddleRow+Offset,:), where Offset is the value of the **Ending row offset** parameter. When all rows are to be included, the following code defines the ending column: y(:,end) = u(:,MiddleColumn+Offset), where Offset is the value of the **Ending column offset** parameter.

- Middle

  When you select this option, the block uses the middle row or column of the input u as the last row or column of the output y. When the number, *X*, of input rows or columns is even, the block defines the middle one as *X*/2 +1. When the number of input rows or columns is odd, the block defines the middle one as ceil(*X*/2).

  When all columns are to be included, the following code defines the ending row: y(end,:) = u(MiddleRow,:). When all rows are to be included, the following code defines the ending column: y(:,end) = u(:,MiddleColumn).

This block supports Simulink virtual buses.

Examples   The ex_submatrix_ref model uses a Submatrix block to extract a 3-by-2 submatrix from the lower-right corner of a 5-by-7 input matrix.

The following figure shows the operation of the Submatrix block with a 5-by-7 input matrix of random integer elements, randi([0 9],5,7).
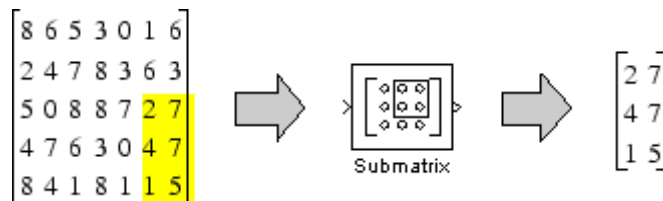
There are often several possible parameter combinations that you can use to select the *same* submatrix from the input. For example, in the case of a 5-by-7 input matrix, instead of specifying Last for **Ending column**, you could select the same submatrix by specifying

- **Ending column** = Index
- **Ending column index** = 7

Function Block Parameters: Submatrix

Submatrix (mask) (link)

Return selected portion of input matrix. 1-D input signals are treated as 2-D column vectors. The output is always 2-D.

This block performs 1-based indexing.

Parameters

Row span: Range of rows

Starting row: First

Ending row: Last

Column span: Range of columns

Starting column: First

Ending column: Last

OK   Cancel   Help   Apply
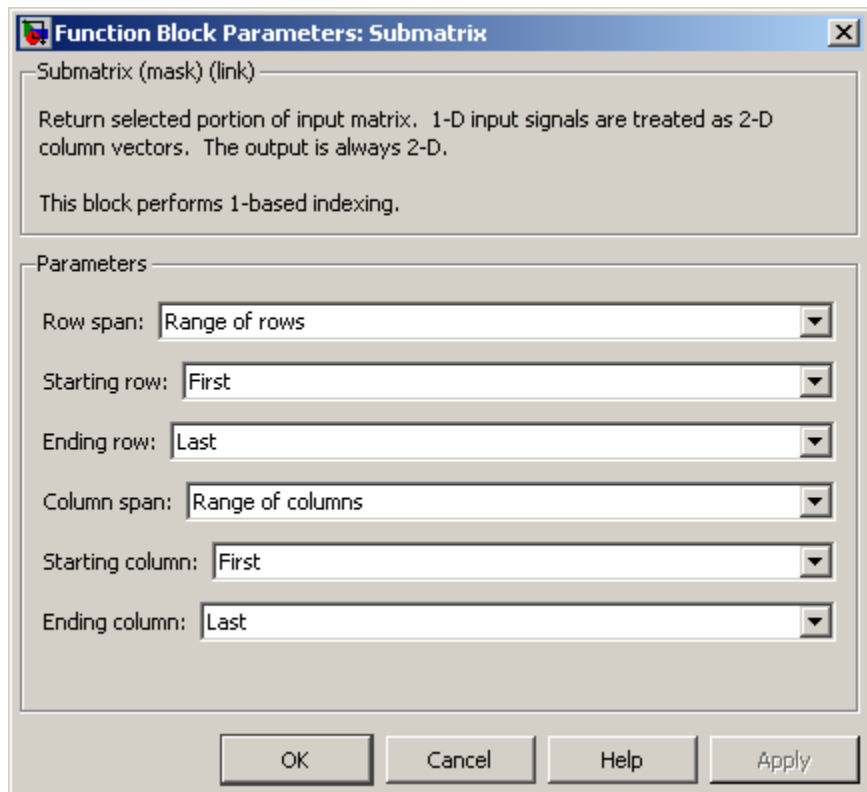
Dialog Box

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.

Submatrix

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 you select `One row` from **Row span**, and **Starting
row** when you select `Range of rows` from **Row span**.

Row index/Starting row index

The index of the input row to be used as the first row of the output.
Row index is enabled when you select `Index` from Row, and
Starting row index when you select `Index` from **Starting row**.

Row offset/Starting row offset

The offset of the input row to be used as the first row of the
output. **Row offset** is enabled when you select `Offset from
middle` or `Offset from last` from **Row**, and Starting row offset
is enabled when you select `Offset from middle` or `Offset from
last` from **Starting row**.

Ending row

The input row to be used as the last row of the output. This
parameter is enabled when you select `Range of rows` from **Row
span** and you select any option but `Last` from **Starting row**.

Ending row index

The index of the input row to be used as the last row of the
output. This parameter is enabled when you select `Index` from
Ending row.

Ending row offset

The offset of the input row to be used as the last row of the output.
This parameter is enabled when you select `Offset from middle`
or `Offset from last` from **Ending row**.

Column span

The range of input columns to be retained in the output. Options
are `All columns`, `One column`, or `Range of columns`.

Column/Starting column
> The input column to be used as the first column of the output. **Column** is enabled when you select One column from **Column span**, and **Starting column** is enabled when you select Range of columns from **Column span**.

Column index/Starting column index
> The index of the input column to be used as the first column of the output. **Column index** is enabled when you select Index from Column, and **Starting column index** is enabled when you select Index from **Starting column**.

Column offset/Starting column offset
> The offset of the input column to be used as the first column of the output. **Column offset** is enabled when you select Offset from middle or Offset from last from Column. **Starting column offset** is enabled when you select Offset from middle or Offset from last from **Starting column**.

Ending column
> The input column to be used as the last column of the output. This parameter is enabled when you select Range of columns from **Column span** and you select any option but Last from **Starting column**.

Ending column index
> The index of the input column to be used as the last column of the output. This parameter is enabled when you select Index from **Ending column**.

Ending column offset
> The offset of the input column to be used as the last column of the output. This parameter is enabled when you select Offset from middle or Offset from last from **Ending column**.

Submatrix

Supported Data Types

| Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Enumerated

See Also

Reshape	Simulink
Selector	Simulink
Variable Selector	DSP System Toolbox
reshape	MATLAB

See "Split Multichannel Signals into Several Multichannel Signals" for related information.

Purpose   Solve *AX=B* using singular value decomposition

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers

dspsolvers
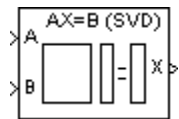
Description   The SVD Solver block solves the linear system *AX=B*, which can be overdetermined, underdetermined, or exactly determined. The system is solved by applying singular value decomposition (SVD) factorization to the *M*-by-*N* matrix *A*, at the A port. The input to the B port is the right side *M*-by-*L* matrix, *B*. The block treats length-*M* unoriented vector input as an *M*-by-1 matrix.

The output at the X port is the *N*-by-*L* matrix, *X*. *X* is chosen to minimize the sum of the squares of the elements of *B-AX* (the residual). When *B* is a vector, this solution minimizes the vector 2-norm of 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 *k*th column of *B*, and X_k is the *k*th column of *X*.

X is known as the minimum-norm-residual solution to *AX=B*. The minimum-norm-residual solution is unique for overdetermined and exactly determined linear systems, but it is not unique for underdetermined linear systems. Thus when the SVD Solver block is applied to an underdetermined system, the output *X* is chosen such that the number of nonzero entries in *X* is minimized.

SVD Solver

Dialog Box
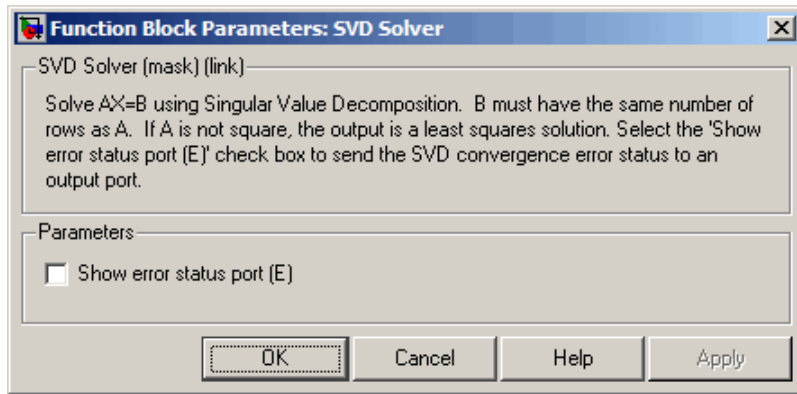
Show error status port

Select to enable the E output port, which reports a failure to converge. The possible values you can receive on the port are:

- 0 — The singular value decomposition calculation converges.

- 1 — The singular value decomposition calculation does not converge.

If the singular value decomposition calculation fails to converge, the output at port X is an undefined matrix of the correct size.

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point
	• Single-precision floating point
B	• Double-precision floating point
	• Single-precision floating point
X	• Double-precision floating point
	• Single-precision floating point
E	• Boolean

See Also

Autocorrelation LPC	DSP System Toolbox
Cholesky Solver	DSP System Toolbox
LDL Solver	DSP System Toolbox
Levinson-Durbin	DSP System Toolbox
LU Inverse	DSP System Toolbox
Pseudoinverse	DSP System Toolbox
QR Solver	DSP System Toolbox
Singular Value Decomposition	DSP System Toolbox

See "Linear System Solvers" for related information.

Time Scope

Purpose	Display time-domain signals
Library	Sinks
	dspsnks4

Description
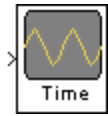
The Time Scope block displays signals in the time domain. The Time Scope block accepts input signals with the following characteristics:

- Continuous or discrete sample time
- Real- or complex-valued
- Fixed or variable size dimensions
- Floating- or fixed-point data type
- N-dimensional
- Simulink enumerations
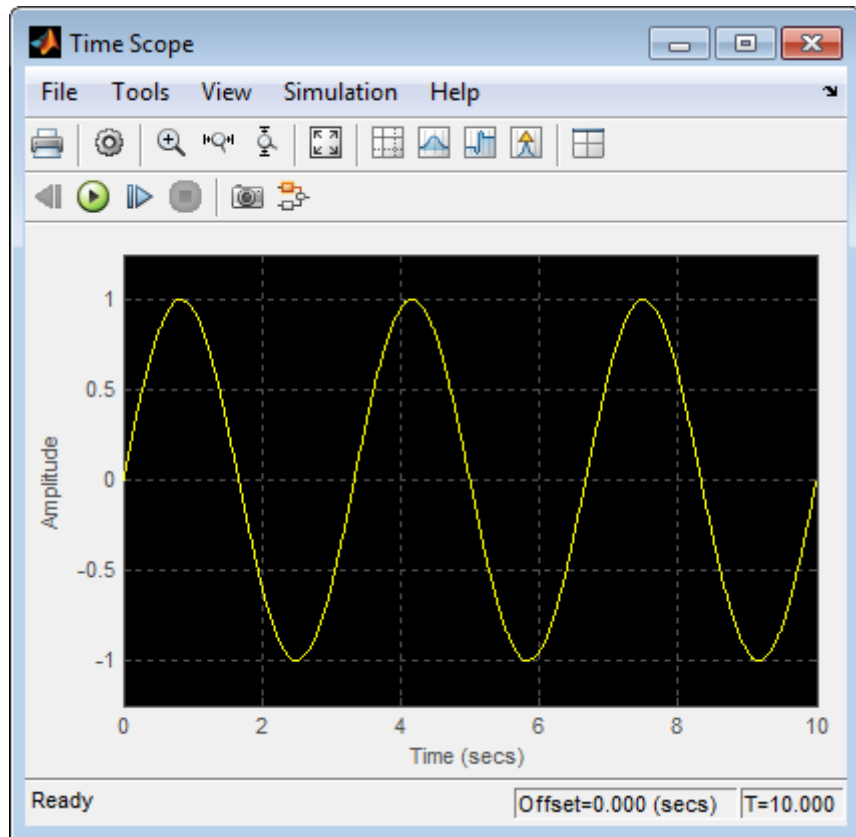
You can use the Time Scope block in models running in Normal or Accelerator simulation modes. You can also use the Time Scope block in models running in Rapid Accelerator or External simulation modes, with some limitations. See the "Supported Simulation Modes" on page 2-1525 section for more information.

You can use the Time Scope block inside of all subsystems and conditional subsystems. *Conditional subsystems* include enabled subsystems, triggered subsystems, enabled and triggered subsystems, and function-call subsystems. See "About Conditional Subsystems" in the Simulink documentation for more information.

Time Scope

For an example that uses the Time Scope block, see the Display Time-Domain Data section in the DSP System Toolbox documentation.

See the following sections for more information on the Time Scope:

Note For information about the Time Scope System object, see `dsp.TimeScope`.

Displaying Multiple Signals

Multiple Signal Input

You can configure the Time Scope block to show multiple signals within the same display or on separate displays. By default, the signals appear as different-colored lines on the same display. The signals can have

different dimensions, sample rates, and data types. Each signal can be either real or complex valued. You can set the number of input ports on the Time Scope block in the following ways:

- Right-click the Time Scope block in your model to bring up the context menu. Point your cursor to the **Signals & Ports > Number of Input Ports** item on the context menu. You can then select the number of input ports for the Time Scope block. If the desired number of input signals is 1, 2, or 3, then click on the appropriate value. To configure a Time Scope block to have more than three input ports, select **More**, and enter a number for the **Number of input ports** parameter.

- Open the Time Scope window by double-clicking the Time Scope block in your model. To specify how many input ports the Time Scope block should have, in the Time Scope menu, select **File > Number of Input Ports** from the scope menu.

An input signal may contain multiple channels, depending on its dimensions. Multiple channels of data are always shown as different-colored lines on the same display.

Multiple Time Offsets

You can offset all channels of an input signal by the same number of seconds or offset each channel independently. To offset all channels equally, select **View > Properties**, and specify a scalar value for the **Time display offset** parameter on the **Main** pane. To offset each channel independently, specify a vector of offset values. When you specify a **Time display offset** vector of length N, the scope offsets the input channels as follows:

- When N is equal to the number of input channels, the scope offsets each channel according to its corresponding value in the offset vector.

- When N is less than the number of input channels, the scope applies the values you specify in the offset vector to the first N input channels. The scope does not offset the remaining channels.

- When *N* is greater than the number of input channels, the scope offsets each input channel according to the corresponding value in the offset vector. The scope ignores all values in the offset vector that do not correspond to a channel of the input.

Multiple Displays

You can display multiple channels of data on different displays in the Scope window. In the Scope toolbar, select **View > Layout**, or select the Layout button ( ). This feature allows you to tile the window into a number of separate displays, up to a grid of 4 rows and 4 columns. For example, if there are three inputs to the Scope, you can display the signals in separate displays by selecting row 3, column 1, as shown in the following figure.

After you select row 3, column 1, the Scope window has three separate displays, as shown in the following figure.
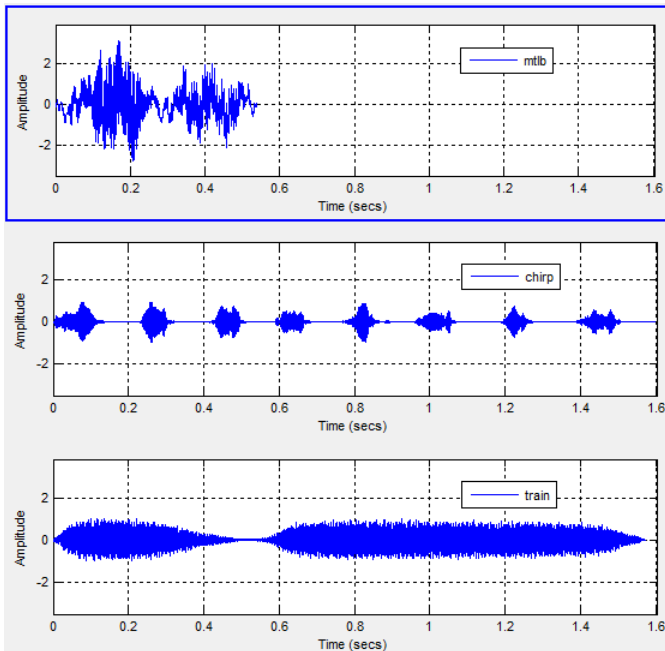
Signal Display

Time Scope uses the simulation start time and stop time in order to determine the default time range. However, you can define the length of simulation time for which the Time Scope displays data. To change the signal display settings, select **View > Properties** to bring up the Properties dialog box. Then, modify the values for the **Time span** and **Time display offset** parameters on the **Time** tab. For example, if you set the **Time span** to 20 seconds and the **Time display offset** to 0, the scope displays 20 seconds' worth of simulation data at a time. The values on the *time*-axis of the Time Scope display remain the same throughout simulation.

To communicate the simulation time that corresponds to the current display, the scope uses the **Time units**, **Time offset**, and **Simulation time** indicators on the scope window. The following figure highlights these and other important aspects of the Time Scope window.

Time Scope

- **Minimum time-axis limit** — The Time Scope sets the minimum *time*-axis limit using the value of the **Time display offset** parameter on the **Time** tab of the Properties dialog box. If you specify a vector of values for the **Time display offset** parameter, the scope uses the smallest of those values to set the minimum *time*-axis limit.

- **Maximum time-axis limit** — The Time Scope sets the maximum *time*-axis limit by summing the value of **Time display offset** parameter with the value of the **Time span** parameter. If you specify a vector of values for the **Time display offset** parameter, the scope sets the maximum *time*-axis limit by summing the largest of those values with the value of the **Time span** parameter.

- **Simulation status** — Provides the current status of the model simulation. The status can be one of the following conditions:

  - `Initializing`

  - `Ready`

  - `Running`

  - `Paused`

  The **Simulation status** is part of the **Status Bar** in the Time Scope window. You can choose to hide or display the entire **Status Bar**. From the Time Scope menu, select **View > Status Bar**.

- **Time units** — The units used to describe the *time*-axis. The Time Scope sets the time units using the value of the **Time Units** parameter on the **Time** tab of the Properties dialog box. By default, this parameter is set to `Metric (based on Time Span)` and displays in metric units such as milliseconds, microseconds, minutes, days, etc. You can change it to `Seconds` to always display the *time*-axis values in units of seconds. You can change it to `None` to not display any units on the *time*-axis. When you set this parameter to `None`, then Time Scope shows only the word `Time` on the *time*-axis.

  To hide both the word `Time` and the values on the *time*-axis, set the **Show time-axis labels** parameter to `None`. To hide both the word `Time` and the values on the *time*-axis in all displays except the bottom ones in each column of displays, set this parameter to `Bottom Displays Only`. This behavior differs from the Simulink Scope block, which always shows the values but never shows a label on the *x*-axis.

- **Time offset** — The `Time offset` value helps you determine the simulation times for which the scope is displaying data. The value is always in the range $0 \leq$ `Time offset` $\leq$ **Simulation time**.

  For example, if you set the **Time span** to `20` seconds, and you see a `Time offset` of `0 (secs)` on the scope window, the scope is displaying data for the first `0` to `20` seconds of simulation time. When you see the `Time offset` change to `20 (secs)`, the scope is displaying data for simulation times is greater than `20` seconds and less than or equal to `40` seconds. The scope continues to update the `Time offset` value until the simulation is complete.

- **Simulation time** — When the model is running or simulation has been paused, the scope displays the current simulation time. If the model simulation completes or is stopped, the scope displays the time at which the simulation stopped. The **Simulation time** is part of the **Status Bar** in the Time Scope window. You can choose to hide or display the entire **Status Bar**. From the Time Scope menu, select **View > Status Bar** .

Axes Maximization

You can specify whether to display the Time Scope in maximized axes mode. In this mode, the axes are expanded to fill the entire display. To conserve space, no labels appear in each display. Instead, tick-mark values appear on top of the plotted data. The following figure highlights the important indicators in this mode of the Time Scope window.
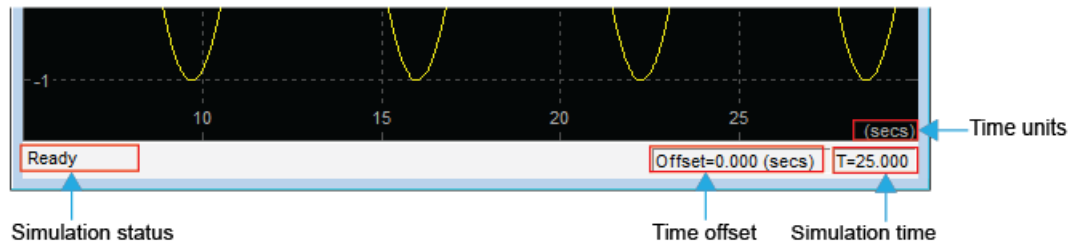
To toggle this mode, in the Time Scope menu, select **View > Properties** to bring up the Properties dialog box. In the **Main** pane, set the **Maximize axes** parameter. You can select one of the following options:

- Auto — In this mode, the axes appear maximized in all displays only if the **Title** and **Y-Axis label** parameters are empty for every display. If you enter any value in any display for either of these parameters, the axes are not maximized.

- On — In this mode, the axes appear maximized in all displays. Any values entered into the **Title** and **Y-Axis label** parameters are hidden.

- Off — In this mode, none of the axes appear maximized.

The default setting is Auto.

Reduce Updates to Improve Performance

By default, the Time Scope updates the displays periodically at a rate not exceeding 20 hertz. If you would like the Time Scope to update on every simulation time step, you can disable the **Reduce Updates to Improve Performance** option. In the Time Scope menu, select

Simulation > Reduce Updates to Improve Performance to clear the check box. However, as a recommended practice, you can leave this option enabled because doing so can significantly improve the speed of the simulation.

Toolbar   The Time Scope toolbar contains the following buttons.

Print and Properties Buttons

Button	Menu Location	Shortcut Keys	Description
🖨	**File > Print**	**Ctrl+P**	Print the current scope window. Printing is available only when the scope display is not changing. You can enable printing by placing the scope in snapshot mode, pausing or stopping model simulation, or simulating the model in step-forward mode. To print the current scope window to a figure rather than sending it to your printer, select **File > Print to figure**.
⚙	**View > Properties**	N/A	Open the **Properties** dialog box. See the "Properties Dialog Box" on page 2-1479 section for more information.

Axes Control Buttons

	Tools > Zoom In	N/A	When this tool is active, you can zoom in on the scope window. To do so, click in the center of your area of interest, or click and drag your cursor to draw a rectangular area of interest inside the scope window.
	Tools > Zoom X	N/A	When this tool is active, you can zoom in on the *x*-axis. To do so, click inside the scope window, or click and drag your cursor along the *x*-axis over your area of interest.
	Tools > Zoom Y	N/A	When this tool is active, you can zoom in on the *y*-axis. To do so, click inside the scope window, or click and drag your cursor along the *y*-axis over your area of interest.
	Tools > Scale Axes Limits	**Ctrl+A**	Click this button to scale the axes in the active scope window. Alternatively, you can enable automatic axes scaling by selecting one of the following options from the **Tools** menu: • **Automatically Scale Axes Limits** — When you select this option, the scope scales the axes as needed during simulation. • **Scale Axes Limits at Stop** — When you select this option, the scope scales the axes each time the simulation is stopped.

Measurements Buttons

⊞	**Tools > Measurements > Cursor Measurements**	N/A	Open or close the **Cursor Measurements** panel. This panel puts screen cursors on all the displays. See the "Cursor Measurements Panel" on page 5-1114 section for more information.
⩘	**Tools > Measurements > Signal Statistics**	N/A	Open or close the **Signal Statistics** panel. This panel displays the maximum, minimum, peak-to-peak difference, mean, median, RMS values of a selected signal, and the times at which the maximum and minimum occur. See the "Signal Statistics Panel" on page 5-1116 section for more information.
⊓	**Tools > Measurements > Bilevel Measurements**	N/A	Open or close the **Bilevel Measurements** panel. This panel displays information about a selected signal's transitions, overshoots or undershoots, and cycles. See the "Bilevel Measurements Panel" on page 5-1118 section for more information.
⟰	**Tools > Measurements > Peak Finder**	N/A	Open or close the **Peak Finder** panel. This panel displays maxima and the times at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified.

			See the "Peak Finder Panel" on page 5-1132 section for more information.

Other Buttons

	Menu Location	Shortcut Keys	Description
⊞	**View > Layout**	N/A	Arrange the layout of displays in the Time Scope. This feature allows you to tile your screen into a number of separate displays, up to a grid of 4 rows and 4 columns. You may find multiple displays useful when the Time Scope takes multiple input signals. The default display is 1 row and 1 column. See the "Multiple Displays" on page 5-1104 section for more information.

You can control whether this toolbar appears in the Time Scope window. From the Time Scope menu, select **View > Toolbar**.

Simulation Toolbar

The Simulation Toolbar contains the following buttons.

Button	Menu Location	Shortcut Keys	Description
	Simulation > Simulation Stepping Options	N/A	Open the **Simulation Stepping Options** dialog box. This button appears only when you have previous stepping disabled. See the "Simulation Stepping Options"

Button	Menu Location	Shortcut Keys	Description
			on page 2-1493 section for more information.
◁|	**Simulation > Previous Step**	N/A	Advance the model simulation backward by one time step. This button appears only when you have previous stepping enabled and the model simulation is paused.
⊙	**Simulation > Run**	**Ctrl+T**, **p**, **Space**	Start the model simulation. This button appears only when the model simulation is stopped.
⊙	**Simulation > Continue**	**p**, **Space**	Continue the model simulation. This button appears only when the model simulation is paused.
⊙	**Simulation > Pause**	**p**, **Space**	Pause the model simulation. This button appears only when the model simulation is running.
▷	**Simulation > Step Forward**	**Right arrow**, **Page Down**	Advance the model simulation forward by one time step. This button starts the model simulation, allows it to run for one time step, and then pauses it again. The scope window then updates with the latest data.
⊡	**Simulation > Stop**	**Ctrl+T**, **s**	Stop the model simulation. This button appears only when the model simulation is running or paused.

Time Scope

Button	Menu Location	Shortcut Keys	Description
	Simulation > Simulink Snapshot	N/A	Take a snapshot of the current scope display. This button temporarily freezes the scope display, while allowing simulation to continue running. To unfreeze the scope display and view the current simulation data, toggle this button to turn off snapshot mode.
	View > Highlight Simulink Block	**Ctrl+L**	Bring the model window forward, and highlight the Time Scope block whose display you are currently viewing. The Time Scope block that corresponds to the active Time Scope window flashes three times in the model.

You can control whether this toolbar appears in the Time Scope window. From the Time Scope menu, select **View > Simulation Toolbar**.

To see a full listing of the shortcut keys for these simulation controls, from the Time Scope menu, select **Help > Keyboard Command Help**.

Measurements Panels

The Measurements panels are the five panels that appear to the right side of the Time Scope GUI. These panels are labeled **Trace selection**, **Signal statistics**, **Bilevel measurements**, and **Peak finder**.

The Time Domain Measurements panels only appear if the **Measurements** tool is enabled in the Time Scope — Tools:Plot Navigation Options dialog box. To open this dialog box, in the Time Scope menu, select **File > Configuration** and click the **Tools** pane. If you disable the tool by clearing the **Enabled** check box, the Time Domain Measurements tools no longer display in the Time Scope GUI. You can reenable the tool at any time by selecting the **Enabled** check

box. See the "Tools — Plot Navigation Options" on page 2-1507 section for more information.

Measurements Panel Buttons

Each of the Measurements panels contains the following buttons that enable you to modify the appearance of the current panel.

Button	Description
禾	Move the current panel to the top. When you are displaying more than one panel, this action moves the current panel above all the other panels.
▼	Collapse the current panel. When you first enable a panel, by default, it displays one or more of its panes. Click this button to hide all of its panes to conserve space. After you click this button, it becomes the expand button ▶.
▶	Expand the current panel. This button appears after you click the collapse button to hide the panes in the current panel. Click this button to display the panes in the current panel and show measurements again. After you click this button, it becomes the collapse button ▼ again.
↗	Undock the current panel. This button lets you move the current panel into a separate window that can be relocated anywhere on your screen. After you click this button, it becomes the dock button ↘ in the new window.
↘	Dock the current panel. This button appears only after you click the undock button. Click this button to put the current panel back into the right side of the Scope window. After you click this button, it becomes the undock button ↗ again.
✕	Close the current panel. This button lets you remove the current panel from the right side of the Scope window.

Some panels have their measurements separated by category into a number of panes. Click the pane expand button ▶ to show each pane

that is hidden in the current panel. Click the pane collapse button ▼ to hide each pane that is shown in the current panel.

Trace Selection Panel

When you use the Scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click on any of the other Measurements panels. Choose the signal name for which you would like to display time domain measurements. See the following figure.

You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools > Measurements > Trace Selection**.

Cursor Measurements Panel

The **Cursor Measurements** panel displays screen cursors. You can choose to hide or display the **Cursor Measurements** panel. In the Scope menu, select **Tools > Measurements > Cursor Measurements**. Alternatively, in the Scope toolbar, click the Cursor Measurements button.

Settings Pane

The **Settings** pane enables you to modify the type of screen cursors used to calculate time and value measurements.

- **Screen Cursors**— Shows screen cursors.

- **Horizontal**— Shows horizontal screen cursors.

- **Vertical**— Shows vertical screen cursors.

- **Waveform Cursors**— Shows cursors that attach to the input signals.

- **Lock Cursor Spacing**— Locks the time difference between the two cursors.

Measurements Pane

The **Measurements** pane shows the time and value measurements.

- **1 |**— Shows or enables you to modify the time or value at cursor number one, or both.

- **2 :**— Shows or enables you to modify the time or value at cursor number two, or both.

- **Δt**— Shows the absolute value of the difference in the times between cursor number one and cursor number two.

- **ΔV**— Shows the absolute value of the difference in signal amplitudes between cursor number one and cursor number two.

- **1/Δt**— Shows the rate, the reciprocal of the absolute value of the difference in the times between cursor number one and cursor number two.

- **ΔV/Δt**— Shows the scope, the ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times between cursors.

Signal Statistics Panel

The **Signal Statistics** panel displays the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal. It also shows the times at which the maximum and minimum values occur. You can choose to hide or display the **Signal Statistics** panel. In the Scope menu, select **Tools > Measurements > Signal Statistics**. Alternatively, in the Scope toolbar, click the Signal Statistics button.

🅽 ▼ Signal Statistics		🡥 ✕
	Value	Time (us)
Max	2.343	22.500
Min	-57.337 m	5.000
Peak to Peak	2.400	
Mean	1.150	
Median	1.152	
RMS	1.628	

Signal Statistics Measurements

The **Signal Statistics** panel shows statistics about the portion of the input signal within the *x*-axis and *y*-axis limits of the active display. The statistics shown are:

- **Max** — Shows the maximum or largest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `max` function reference.

- **Min** — Shows the minimum or smallest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `min` function reference.

- **Peak to Peak** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `peak2peak` function reference.

- **Mean** — Shows the average or mean of all the values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `mean` function reference.

- **Median** — Shows the median value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `median` function reference.

- **RMS** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `rms` function reference.

When you use the zoom options in the Scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Bilevel Measurements Panel

The **Bilevel Measurements** panel shows information about a selected signal's transitions, overshoots or undershoots, and cycles. You can choose to hide or display the **Bilevel Measurements** panel. In the Scope menu, select **Tools > Measurements > Bilevel Measurements**. Alternatively, in the Scope toolbar, you can select the Bilevel Measurements button.

The **Bilevel Measurements** panel is separated into four panes, labeled **Settings**, **Transitions**, **Overshoots / Undershoots**, and **Cycles**. You can expand each pane to see the available options.

Settings Pane

The **Settings** pane enables you to modify the properties used to calculate various measurements involving transitions, overshoots, undershoots, and cycles. You can modify the high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level, as shown in the following figure.

- **Auto State Level** — When this check box is selected, the Bilevel measurements panel autodetects the high- and low- state levels of a bilevel waveform. For more information on the algorithm this option uses, see the Signal Processing Toolbox `statelevels` function

reference. When this check box is cleared, you may enter in values for the high- and low- state levels manually.

- **High** — Used to manually specify the value that denotes a positive polarity, or high-state level, as shown in the following figure.

Positive Polarity Pulse

- **Low** — Used to manually specify the value that denotes a negative polarity, or low-state level, as shown in the following figure.

Negative Polarity Pulse

- **State Level Tolerance** — Tolerance within which the initial and final levels of each transition must be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.

- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.

- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low- state levels. In the following figure, the mid-reference level is shown as the horizontal line, and its corresponding mid-reference level instant is shown as the vertical line.

- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.

- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs used for computing a valid settling time. This value is equivalent to the input parameter, D, which you can set when you run the settlingtime function. The settling time is displayed in the **Overshoots/Undershoots** pane.

Transitions Pane

The **Transitions** pane displays calculated measurements associated with the input signal changing between its two possible state level values, high and low.

A positive-going transition, or *rising edge*, in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-going transition has a slope value greater than zero. The following figure shows a positive-going transition.

Whenever there is a plus sign (+) next to a text label, this symbol refers to measurement associated with a rising edge, a transition from a low-state level to a high-state level.

A negative-going transition, or falling edge, in a bilevel waveform is a transition from the high-state level to the low-state level. A negative-going transition has a slope value less than zero. The following figure shows a negative-going transition.

$$\frac{dX}{dt} < 0$$

Whenever there is a minus sign (–) next to a text label, this symbol refers to measurement associated with a falling edge, a transition from a high-state level to a low-state level.

The Transition measurements assume that the amplitude of the input signal is in units of volts. You must convert all input signals to volts for the Transition measurements to be valid.

- **High** — The high-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals:Time Domain Options dialog box. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.

- **Low** — The low-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals:Time Domain Options dialog box. For

more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.

- **Amplitude** — Difference in amplitude between the high-state level and the low-state level.

- **+ Edges** — Total number of positive-polarity, or rising, edges counted within the displayed portion of the input signal.

- **+ Rise Time** — Average amount of time required for each rising edge to cross from the lower-reference level to the upper-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `risetime` function reference.

- **+ Slew Rate** — Average slope of each rising-edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. The region in which the slew rate is calculated appears in gray in the following figure.

For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.

- – **Edges** — Total number of negative-polarity or falling edges counted within the displayed portion of the input signal.

- – **Fall Time** — Average amount of time required for each falling edge to cross from the upper-reference level to the lower-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `falltime` function reference.

- – **Slew Rate** — Average slope of each falling edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. For more information on the algorithm

this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.

Overshoots/Undershoots

The **Overshoots/Undershoots** pane displays calculated measurements involving the distortion and damping of the input signal. *Overshoot* and *undershoot* refer to the amount that a signal respectively exceeds and falls below its final steady-state value. *Preshoot* refers to the amount prior to a transition that a signal varies from its initial steady-state value. This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.

The next figure shows preshoot, overshoot, and undershoot for a falling-edge transition.

- **+ Preshoot** — Average lowest aberration in the region immediately preceding each rising transition.

- **+ Overshoot** — Average highest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.

- **+ Undershoot** — Average lowest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.

- **+ Settling Time** — Average time required for each rising edge to enter and remain within the tolerance of the high-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the high-state level. This crossing is illustrated in the following figure.

You can modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

- – **Preshoot** — Average highest aberration in the region immediately preceding each falling transition.

- – **Overshoot** — Average highest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.

- – **Undershoot** — Average lowest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.

- – **Settling Time** — Average time required for each falling edge to enter and remain within the tolerance of the low-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the low-state level. You can

modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

Cycles

The **Cycles** pane displays calculated measurements pertaining to repetitions or trends in the displayed portion of the input signal.

- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. The Bilevel measurements panel calculates period as follows. It takes the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition. These mid-reference level instants appear as red dots in the following figure.

For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulseperiod` function reference.

- **Frequency** — Reciprocal of the average period. Whereas period is typically measured in some metric form of seconds, or seconds per cycle, frequency is typically measured in hertz or cycles per second.

- **+ Pulses** — Number of positive-polarity pulses counted.

- **+ Width** — Average duration between rising and falling edges of each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.

- **+ Duty Cycle** — Average ratio of pulse width to pulse period for each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.

- **– Pulses** — Number of negative-polarity pulses counted.

- **– Width** — Average duration between rising and falling edges of each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.

- **– Duty Cycle** — Average ratio of pulse width to pulse period for each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.

When you use the zoom options in the Scope, the bilevel measurements automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the x-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one rising edge to make the **Bilevel Measurements** panel display information about only that

particular rising edge. However, this feature does not apply to the **High** and **Low** measurements.

Peak Finder Panel

The **Peak Finder** panel displays maxima and the times at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified. You can choose to hide or display the **Peak Finder** panel. In the Scope menu, select **Tools > Measurements > Peak Finder**. Alternatively, in the Scope toolbar, select the Peak Finder ⬛ button.

The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

Settings Pane

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input

signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.

- **Max Num of Peaks** — The maximum number of peaks to show. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.

- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.

- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples. Peak excursion is illustrated alongside peak threshold in the following figure.

Whereas the peak threshold is a minimum value necessary for a sample value to be a peak, the peak excursion is the minimum difference between a peak sample and the samples to its left and right

in the time domain. In the figure, the green vertical line illustrates the lesser of the two height differences between the labeled peak and its neighboring samples. This height difference must be greater than the **Peak Excursion** value for the labeled peak to be classified as a peak. Compare this setting to peak threshold, which is illustrated by the red horizontal line. The amplitude must be above this horizontal line for the labeled peak to be classified as a peak.

The peak excursion setting is equivalent to the THRESHOLD parameter, which you can set when you run the findpeaks function.

- **Label peaks** — When this check box is selected, the Peak Finder displays the calculated peak values directly on the plot. By default, this check box is cleared and no values are displayed on the plot.

- **Sort by time** — When this check box is selected, the Peak Finder puts the largest calculated peak values in the time range shown in the display in time-sorted order. The Peak Finder displays the peak values in the **Peaks** pane. The **Max Num of Peaks** parameter still controls the number of peaks listed. By default, this check box is cleared and the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height.

Peaks Pane

The **Peaks** pane displays all of the largest calculated peak values and the times at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list. Using the **Sort by time** check box, you set the order in which the peaks are listed.

The numerical values displayed in the **Value** column are equivalent to the pks output argument returned when you run the findpeaks function. The numerical values displayed in the **Time** column are similar to the locs output argument returned when you run the findpeaks function.

When you use the zoom options in the Scope, the peak locations automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the

x-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one peak to make the **Peak Finder** panel display information about only that particular peak.

Properties Dialog Box

The Properties dialog box controls various properties about the Time Scope displays. From the Time Scope menu, select **View > Properties** to open this dialog box. Alternatively, in the Time Scope toolbar, click the Properties ⚙ button.

Main Pane

The **Main** pane of the Properties dialog box appears as follows.
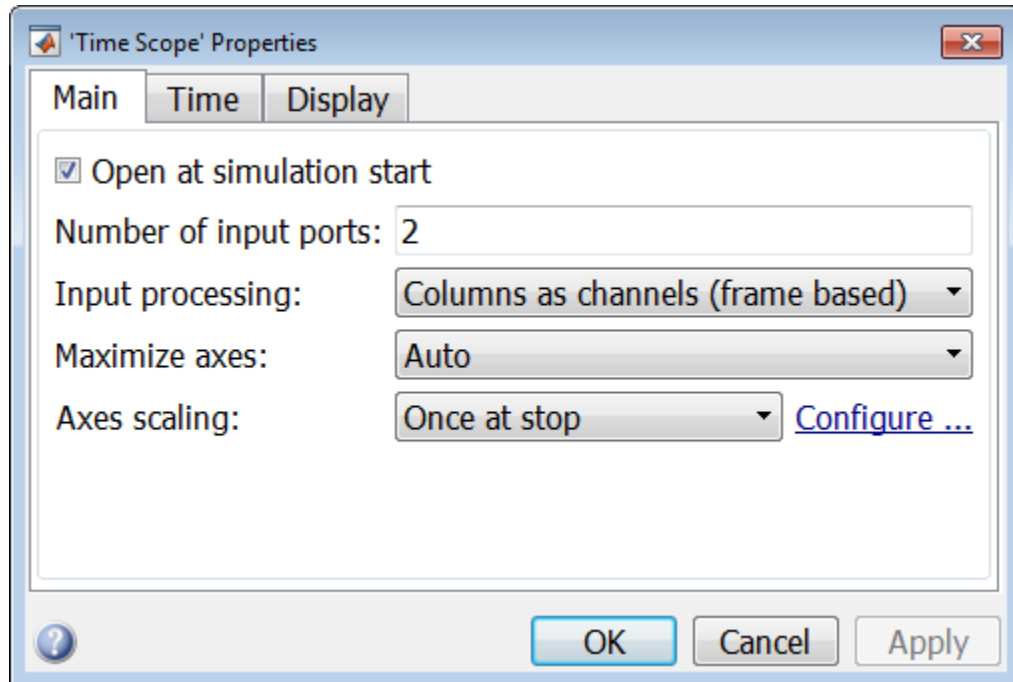
Time Scope

Open at simulation start

Select this check box to ensure that the Time Scope opens when the simulation starts.

Number of input ports

Specify the number of input ports that should appear on the left side of the Time Scope block.

Input processing

Specify whether the Time Scope should treat the input signal as `Columns as channels (frame based)` or `Elements as channels (sample based)`. Tunable.

Frame-based processing is only available for discrete input signals. For more information about frame-based input channels, see the "What Is Frame-Based Processing?" section in the DSP System Toolbox documentation. For an example that uses the Time Scope block and frame-based input signals, see the Display Time-Domain Data in the Time Scope section in the DSP System Toolbox documentation.

Maximize axes

Specify whether to display the Time Scope in maximized axes mode. In this mode, each of the axes are expanded to fit into the entire display. To conserve space, no labels appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- `Auto` — In this mode, the axes appear maximized in all displays only if the **Title** and **Y-Axis label** parameters are empty for every display. If you enter any value in any display for either of these parameters, the axes are not maximized.

- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the **Title** and **Y-Axis label** parameters are hidden.

- `Off` — In this mode, none of the axes appear maximized.

The default setting is `Auto`.

Axes scaling

Specify when the Time Scope should automatically scale the axes. You can select one of the following options:

- Manual — When you select this option, the Time Scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:

  - Select **Tools > Scale axes limits**.
  - Press the **Scale Axes Limits** toolbar button.
  - When the scope GUI is the active window, press **Ctrl** and **A** simultaneously.

- Auto — When you select this option, the Time Scope scales the axes as needed, both during and after simulation. Selecting this option enables the **Do not allow Y-axis limits to shrink** check box. By default, this parameter is selected, and the Time Scope does not shrink the *y*-axis limits when scaling the axes.

- Once at stop — Selecting this option causes the Time Scope to scale the axes each time simulation is stopped. The Time Scope does not scale the axes during simulation.

This parameter is Tunable.

Note Click the link labeled **Configure** to the right of the **Axes scaling** parameter to see additional axes scaling parameters. After you click this button, its label changes to **Hide**. To hide these additional parameters, click the **Hide** link.

Do not allow Y-axis limits to shrink

When you select this parameter, the *y*-axis limits are only allowed to grow during axes scaling operations. If you clear this check box, the *y*-axis limits may shrink during axes scaling operations.

This parameter appears only when you select `Auto` for the **Axis Scaling** parameter. When you set the **Axis Scaling** parameter to `Manual` or `Once at stop`, the y-axis limits are allowed to shrink. Tunable.

Y-axis Data range (%)

Set the percentage of the y-axis the Scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to `100`, the Scope scales the y-axis limits such that your data uses the entire y-axis range. If you then set this parameter to `30`, the Scope increases the y-axis range such that your data uses only 30% of the y-axis range. Tunable.

Y-axis Align

Specify where the Scope should align your data with respect to the y-axis when it scales the axes. You can select `Top`, `Center`, or `Bottom`. Tunable.

Scale X-axis limits

Check this box to allow the Scope to scale the x-axis limits when it scales the axes. Tunable.

X-axis Data range (%)

Set the percentage of the x-axis the Scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to `100`, the Scope scales the x-axis limits such that your data uses the entire x-axis range. If you then set this parameter to `30`, the Scope increases the x-axis range such that your data uses only 30% of the x-axis range. Use the x-axis **Align** parameter to specify data placement with respect to the x-axis.

This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

X-axis Align

Specify how the Scope should align your data with respect to the x-axis: `Left`, `Center`, or `Right`. This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

Time Pane

The **Time** pane of the Properties dialog box appears as follows.
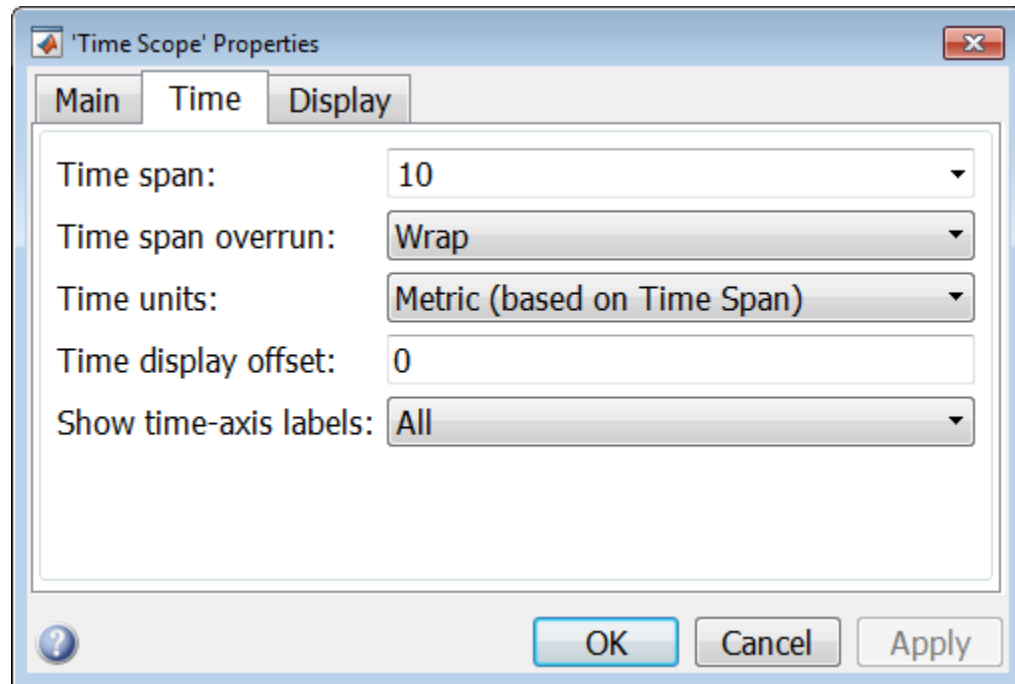
Time span

Specify the time span, either by selecting a predefined option or by entering a numeric value in seconds. You can select one of the following options:

- Auto calculate — In this mode, the Time Scope automatically calculates the appropriate value for time span from the difference between the simulation "Start time" and "Stop time" parameters.

- One frame period — In this mode, the Time Scope uses the frame period of the input signal to the Time Scope block. This option is only available when the **Input processing** parameter is set to Columns as channels (frame based). This option is not available when you set the **Input processing** parameter to Elements as channels (sample based).

- `<user defined>` — In this mode, you specify the time span by replacing the text `<user defined>` with a numeric value in seconds.

The scope sets the *time*-axis limits using the value of this parameter and the value of the **Time display offset** parameter. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the *time*-axis limits as shown in the following figure.

Tunable. For more information, see "Signal Display" on page 2-1447.

Time span overrun

Specify how the scope displays new data beyond the visible time span. The default setting is `Wrap`. You can select one of the following options:

- `Wrap` — In this mode, the scope displays new data until the data reaches the maximum *time*-axis limit. When the data reaches the maximum *time*-axis limit of the scope window, the Time Scope clears the display. The Time Scope then updates the time offset value and begins displaying subsequent data points starting from the minimum *time*-axis limit.

- `Scroll` — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance, but it is beneficial for debugging and monitoring time-varying signals.

This parameter is Tunable.

Time units

Specify the units used to describe the *time*-axis. The default setting is Metric. You can select one of the following options:

- Metric — In this mode, Time Scope converts the times on the *time*-axis to some metric units such as milliseconds, microseconds, nanoseconds, etc. Time Scope chooses the appropriate metric units, based on the minimum *time*-axis limit and the maximum *time*-axis limit of the scope window.

- Seconds — In this mode, Time Scope always displays the units on the *time*-axis as seconds.

- None — In this mode, Time Scope displays no units on the *time*-axis. Time Scope shows only the word Time on the *time*-axis.

Tunable.

Time display offset

This parameter allows you to offset the values displayed on the *time*-axis by a specified number of seconds. When you specify a scalar value, the scope offsets all channels equally. When you specify a vector of offset values, the scope offsets each channel independently. Tunable.

When you specify a **Time display offset** vector of length *N*, the scope offsets the input channels as follows:

- When *N* is equal to the number of input channels, the scope offsets each channel according to its corresponding value in the offset vector.

- When *N* is less than the number of input channels, the scope applies the values you specify in the offset vector to the first *N* input channels. The scope does not offset the remaining channels.

- When *N* is greater than the number of input channels, the scope offsets each input channel according to the corresponding value in the offset vector. The scope ignores all values in the offset vector that do not correspond to a channel of the input.

Time Scope

The scope computes the *time*-axis range using the values of the **Time display offset** and **Time span** parameters. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the *time*-axis limits as shown in the following figure.

Similarly, when you specify a vector of values, the scope sets the minimum *time*-axis limit using the smallest value in the vector. To set the maximum *time*-axis limit, the scope sums the largest value in the vector with the value of the **Time span** parameter. For more information, see "Signal Display" on page 2-1447.

Show time-axis labels

Specify how to display the time units used to describe the *time*-axis. The default setting is `All`. You can select one of the following options:

- `All` — In this mode, the *time*-axis labels appear in all displays.

- `None` — In this mode, the *time*-axis labels do not appear in the displays.

- `Bottom Displays Only` — In this mode, the *time*-axis labels appear in only the bottom row of the displays.

Tunable.

Display Pane

The **Display** pane of the Properties dialog box appears as follows.

Select Display

Specify the active display as a number, where a display number corresponds to the index of the input signal. The index of a display is its column-wise placement index. The default setting is 1. Set this parameter to control the display for which you intend to modify the title, legend, grid, *y*-axis limits, *y*-axis label, and magnitude/phase display parameters. Tunable.

Title

Specify the active display title as a string. By default, the active display has no title. Tunable.

Time Scope

Show legend

Select this check box to show the legend in the active display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn the legend off, clear the **Show legend** check box. Tunable.

You can edit the name of any channel in the legend. To do so, double-click the current name, and enter a new channel name. By default, the scope names each channel according to either its signal name or the name of the block from which it comes. If the signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal named MySignal would have the following default names in the channel legend: MySignal:1, MySignal:2.

The Time Scope does not display signal names that were labeled within an unmasked subsystem. You must label all input signals to the Time Scope block that originate from an unmasked subsystem.

To change the appearance of any channel of any input signal in the Time Scope window, from the Time Scope menu, select **View > Style**.

Show grid

When you select this check box, a grid appears in the active display of the Time Scope GUI. To hide the grid, clear this check box. Tunable.

Plot signal(s) as magnitude and phase

When you select this check box, the Time Scope splits the magnitude and phase of the input signal on two separate axes within the same active display. By default, this box is cleared. If the input is a complex-valued signal, then the Time Scope plots the real and imaginary portions on the same display. These real and imaginary portions appear as different-colored lines on the same axes within the same active display, as shown in the following figure.
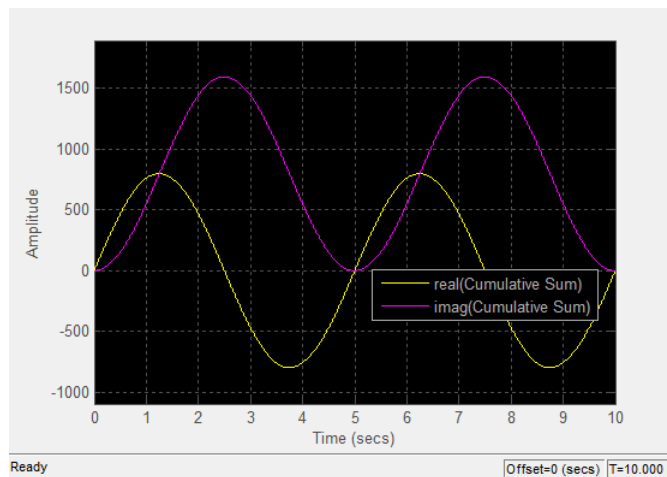
After you select this check box and click **OK**, the active display changes to show the magnitude of the input signal on the top axes and its phase, in degrees, on the bottom axes, as shown in the following figure.
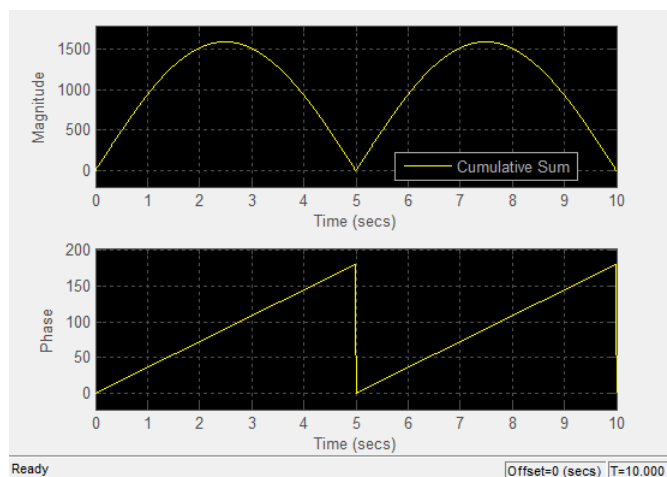
This feature is particularly useful for complex-valued input signals. If the input is a real-valued signal, after you select this check box,

the phase will be 0 degrees when the amplitude of the input signal is nonnegative and 180 degrees when the input signal is negative. Tunable.

Minimum Y-limit

Specify the minimum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a minimum value of –180 degrees.

Maximum Y-limit

Specify the maximum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a maximum value of 180 degrees.

Y-axis label

Specify the text for the scope to display to the left of the *y*-axis. Tunable.

This parameter becomes invisible when you select the **Plot signal(s) as magnitude and phase** check box. When you enable that parameter, the *y*-axis label always appears as Magnitude on the top axes and Phase on the bottom axes.

Style Dialog Box

In the **Style** dialog box, you can customize the style of displays. You are able to change the color of the figure containing the displays, the background and foreground colors of display axes, and properties of lines in a display. From the Scope menu, select **View > Style** to open this dialog box.

Properties

The **Style** dialog box allows you to modify the following properties of the Scope GUI:

Figure color

Specify the color that you want to apply to the background of the Scope GUI. By default, the figure color is gray.

Plot type

Specify the type of plot to use. The default setting is Line. Valid values for **Plot type** are:

- Line — Displays input signal as lines connecting each of the sampled values. This approach is similar to the functionality of the MATLAB line or plot function.

- Stairs — Displays input signal as a *stairstep* graph. A stairstep graph is made up of only horizontal lines and vertical lines. Each

horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. This approach is equivalent to the MATLAB `stairs` function. Stairstep graphs are useful for drawing time history graphs of digitally sampled data.

- `Auto` — Displays input signal as a line graph if it is a continuous signal and displays input signal as a stairstep graph if it is a discrete signal.

This parameter is Tunable.

Select display

Specify the active display as a number, where a display number corresponds to the index of the input signal. The number of a display corresponds to its column-wise placement index. The default setting is `1`. Set this parameter to control which display should have its axes colors, line properties, marker properties, and visibility changed. Tunable.

Axes colors

Specify the color that you want to apply to the background of the axes for the active display.

Properties for line

Specify the signal for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected signal on the active display should be visible. If you clear this check box, the line disappears.

Line

Specify the line style, line width, and line color for the selected signal on the active display.

Marker

Specify marks for the selected signal on the active display to show at data points. This parameter is similar to the `Marker` property for the MATLAB Handle Graphics® plot objects. You can choose any of the marker symbols from the following table.

Specifier	Marker Type
none	No marker (default)
○	Circle
□	Square
×	Cross
•	Point
+	Plus sign
✳	Asterisk
◇	Diamond
▽	Downward-pointing triangle
△	Upward-pointing triangle
◁	Left-pointing triangle
▷	Right-pointing triangle
☆	Five-pointed star (pentagram)
✡	Six-pointed star (hexagram)

Simulation Stepping Options

The Simulation Stepping Options dialog box lets you control the simulation behavior. You can pause the simulation at a specified time, enable stepping back, or specify options for stepping back. You can also modify the number of steps by which to step forward or backward. To open this dialog box, in the Time Scope menu, select **Simulation > Stepping Options** to open this dialog box.

Alternatively, if stepping back is disabled, in the Time Scope toolbar,
click the previous step 🔄 button.
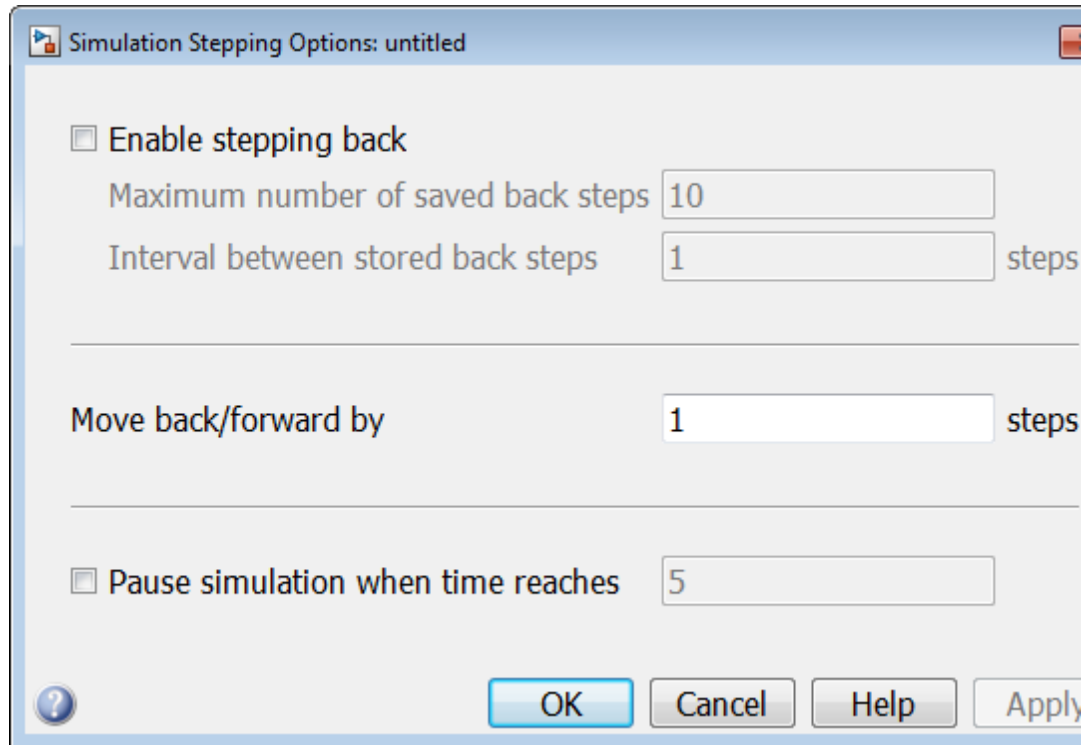
The Simulation Stepping Options dialog box is not unique to Time
Scope; it can also be launched from any Simulink model. To open this
dialog box from any Simulink model, select **Simulation > Stepping
Options**. For more information, see "Simulation Stepping" in the
Simulink documentation.

Enable stepping back

Select this check box to enable the Time Scope to take steps back in

time. When selected, Time Scope enables the previous step button (◁❙)
on the simulation toolbar.

Maximum number of saved back steps

Specify the maximum number of back steps that the Time Scope saves
in memory. To maximize simulation speed, the value for this parameter
should be kept small. The default setting is 10.

Interval between stored back steps

Specify the number of steps between back steps that the Time Scope
saves in memory for previous stepping. Set this parameter to a larger
number to increase the time span of a back step without increasing the
amount of memory used. The default setting is 1.

Move back/forward by

Specify the number of steps forward or backward that the Time Scope

progresses when you click the next step (❙▷) and previous step (◁❙)
buttons. The default setting is 1.

Pause simulation when time reaches

Select this check box to enable the Time Scope to pause the simulation
when it reaches a specified time.

Pause simulation when time reaches

Specify the time at which you want Time Scope to pause when the
check box is selected.

**Core —
General
UI Options**

The Core:General UI Options dialog box controls the general settings
for the Time Scope GUI. From the Time Scope menu bar, select
File > Configuration to open the Time Scope — Configuration dialog
box. Click on the **Core** pane, and the following dialog box appears.
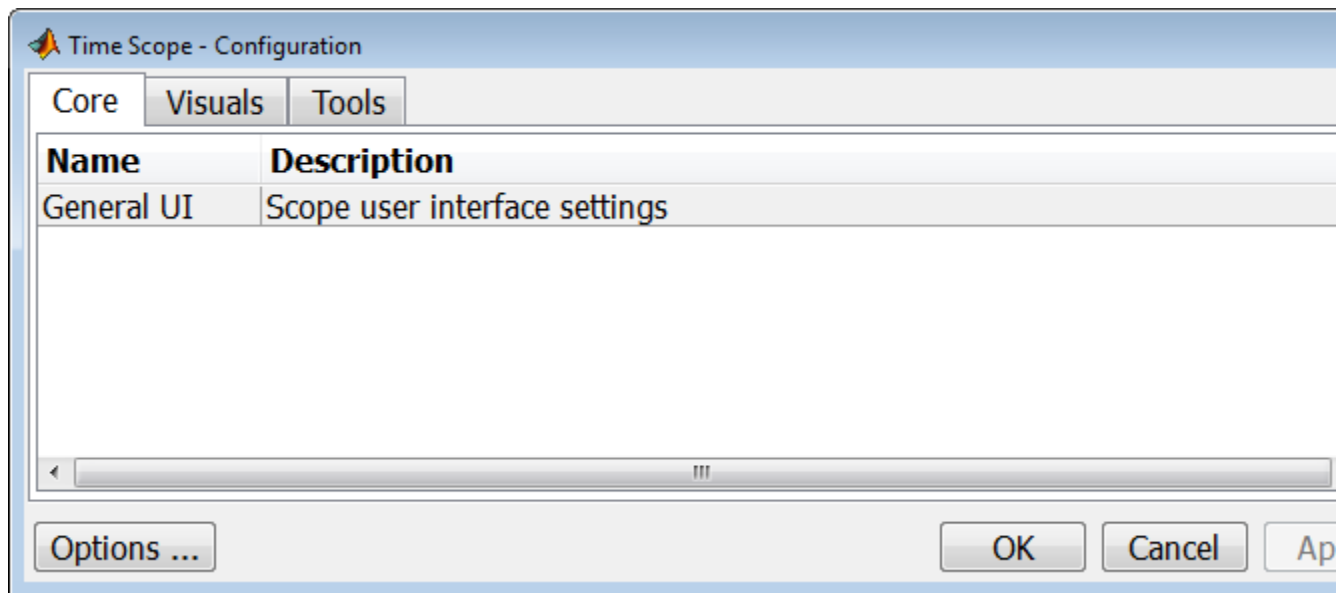
Time Scope

To open the Core:General UI Options dialog box, select **General UI** and click **Options**. The following dialog box appears.

General UI Options
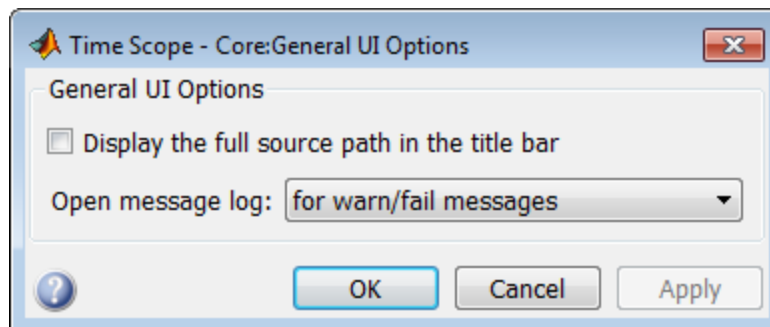
Display the full source path in the title bar

If you select this check box, the Time Scope GUI displays the model name and full Simulink path to the data source in the title bar. Otherwise, a shortened name appears.

Open message log

Use this parameter to control when the Message Log window opens. The Message Log window helps you debug any issues with the Time Scope. Using the drop-down list, you can choose to open the Message Log window under any of the following conditions:

- `for any new messages`

- `for warn/fail messages`

- `only for fail messages`

- `manually`

You can open the Message Log at any time by selecting **Help > Message Log**. The Message Log dialog box provides a system-level record of loaded configuration settings and registered extensions. The Message Log displays summaries and details of each message, and you can filter the display of messages by **Type** and **Category**.

- The **Type** parameter allows you to select which types of messages to display in the Message Log. You can select `All`, `Info`, `Warn`, or `Fail`.

- The **Category** parameter allows you to select the category of messages to display in the Message Log. You can select `All`, `Configuration`, or `Extension`.

  - The scope uses `Configuration` messages to indicate when new configuration files are loaded.

  - The scope uses `Extension` messages to indicate when components are registered. For example, you may see a `Simulink registered` message, indicating that a component has been successfully registered with Simulink and is available for configuration.

Time Scope

Visuals — Time Domain Options

The Time Scope — Visuals:Time Domain Options dialog box controls the visual configuration settings of the Time Scope displays. You can open this dialog box from the Time Scope — Configuration dialog box. From the Time Scope menu bar, select **File > Configuration** to open the Time Scope — Configuration dialog box. Click the **Visuals** pane, and the following dialog box appears.
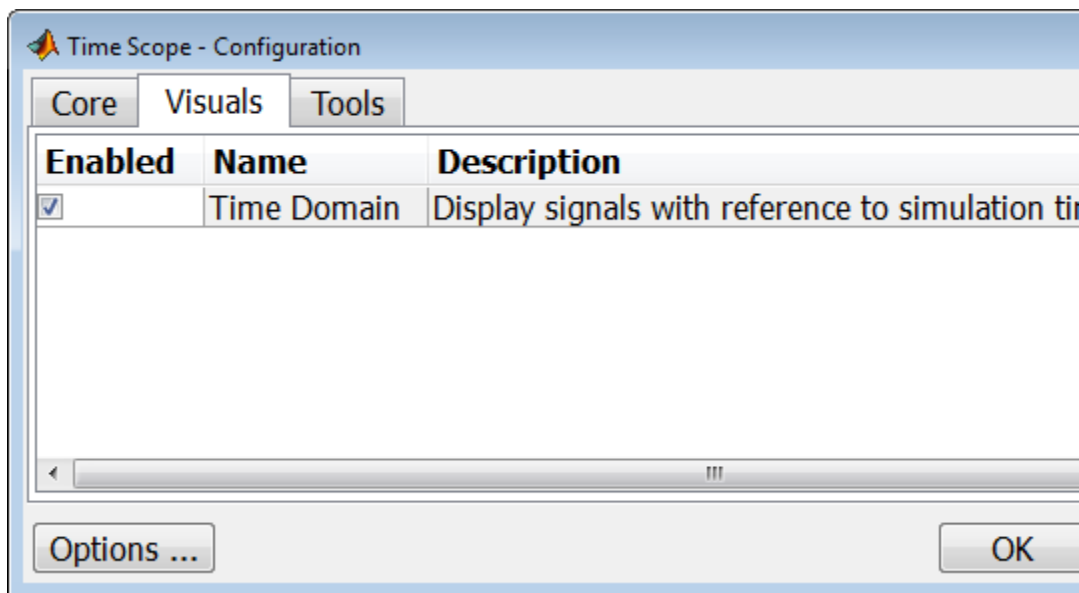
The Time Scope GUI has only one available visual, so you must ensure that the **Enabled** check box is selected. To open the Time Scope — Visuals:Time Domain Options dialog box, select **Time Domain** and click **Options**.

Main Pane

The **Main** pane of the Time Scope — Visuals:Time Domain Options dialog box appears as follows.

Input processing

Specify whether the Time Scope should treat the input signal as
`Columns as channels (frame based)` or `Elements as channels
(sample based)`. Tunable.

Frame-based processing is only available for discrete input signals. For
more information about frame-based input channels, see the "What Is
Frame-Based Processing?" section in the *DSP System Toolbox User's
Guide*. For an example that uses the Time Scope block and frame-based
input signals, see the Display Time-Domain Data in the Time Scope
section in the *DSP System Toolbox User's Guide*.

Time span

Specify the time span, either by selecting a predefined option or by
entering a numeric value in seconds. You can select one of the following
options:

Time Scope

- `Auto calculate` — In this mode, the Time Scope automatically calculates the appropriate value for time span from the difference between the simulation "Start time" and "Stop time" parameters. This option is available only for the Time Scope block but not for the `dsp.TimeScope` System object.

- `One frame period` — In this mode, the Time Scope uses the frame period of the input signal to the Time Scope block. This option is only available when the **Input processing** parameter is set to `Columns as channels (frame based)`. This option is not available when you set the **Input processing** parameter to `Elements as channels (sample based)`.

- `<user defined>` — In this mode, you specify the time span by replacing the text `<user defined>` with a numeric value in seconds.

The scope sets the *time*-axis limits using the value of this parameter and the value of the **Time display offset** parameter. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the *time*-axis limits as shown in the following figure.

This parameter is Tunable.

Time span overrun

Specify how the scope displays new data beyond the visible time span. The default setting is `Wrap`. You can select one of the following options:

- `Wrap` — In this mode, the scope displays new data until the data reaches the maximum *time*-axis limit. When the data reaches the maximum *time*-axis limit of the scope window, the Time Scope clears the display. The Time Scope then updates the time offset value, and begins displaying subsequent data points starting from the minimum *time*-axis limit.

- `Scroll` — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance, but it is beneficial for debugging and monitoring time-varying signals.

This parameter is Tunable.

Time units

Specify the units used to describe the *time*-axis. The default setting is `Metric`. You can select one of the following options:

- `Metric` — In this mode, Time Scope converts the times on the *time*-axis to some metric units such as milliseconds, microseconds, days, etc. Time Scope chooses the appropriate metric units, based on the minimum *time*-axis limit and the maximum *time*-axis limit of the scope window.

- `Seconds` — In this mode, Time Scope always displays the units on the *time*-axis as seconds.

- `None` — In this mode, Time Scope displays no units on the *time*-axis. Time Scope shows only the word `Time` on the *time*-axis.

This parameter is Tunable.

Time display offset

This parameter allows you to offset the values displayed on the *time*-axis by a specified number of seconds. When you specify a scalar value, the scope offsets all channels equally. When you specify a vector of offset values, the scope offsets each channel independently. Tunable.

When you specify a **Time display offset** vector of length *N*, the scope offsets the input channels as follows:

- When *N* is equal to the number of input channels, the scope offsets each channel according to its corresponding value in the offset vector.

- When *N* is less than the number of input channels, the scope applies the values you specify in the offset vector to the first *N* input channels. The scope does not offset the remaining channels.

- When *N* is greater than the number of input channels, the scope offsets each input channel according to the corresponding value in the offset vector. The scope ignores all values in the offset vector that do not correspond to a channel of the input.

The scope computes the *time*-axis range using the values of the **Time display offset** and **Time span** parameters. For example, if you set the **Time display offset** to 5e-6 and the **Time span** to 25e-6, the scope sets the *time*-axis limits as shown in the following figure.

Similarly, when you specify a vector of values, the scope sets the minimum *time*-axis limit using the smallest value in the vector. To set the maximum *time*-axis limit, the scope sums the largest value in the vector with the value of the **Time span** parameter. For more information, see "Signal Display" on page 2-1447.

Show time-axis labels

Specify how to display the time units used to describe the *time*-axis. The default setting is All. You can select one of the following options:

- All — In this mode, the *time*-axis labels appear in all displays.

- None — In this mode, the *time*-axis labels do not appear in the displays.

- Bottom Displays Only — In this mode, the *time*-axis labels appear in only the bottom row of the displays.

Tunable.

Maximize axes

Specify whether to display the Time Scope in maximized axes mode. In this mode, each of the axes are expanded to fit into the entire display. In each display, there is no space to show labels. Tick mark values are shown on top of the plotted data. The default setting is Auto. You can select one of the following options:

- Auto — In this mode, the axes appear maximized in all displays only if the **Title** and **Y-Axis label** parameters are empty for every display. If you enter any value in any display for either of these parameters, the axes are not maximized.

- On — In this mode, the axes appear maximized in all displays. Any values entered into the **Title** and **Y-Axis label** parameters are hidden.

- Off — In this mode, none of the axes appear maximized.

Display Pane

The **Display** pane of the Visuals:Time Domain Options dialog box appears as follows.

Select display

Specify the active display as a number, where a display number corresponds to the index of the input signal. The index of a display is its column-wise placement index. The default setting is 1. Set this parameter to control the display for which you intend to modify the title, legend, grid, y-axis limits, y-axis label, and magnitude/phase display parameters. Tunable.

Title

Specify the active display title as a string. By default, the active display has no title. Tunable.

Show legend

Select this check box to show the legend in the active display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn the legend off, clear the **Show legend** check box. Tunable.

You can edit the name of any channel in the legend. To do so, double-click the current name, and enter a new channel name. By default, the scope names each channel according to either its signal name or the name of the block from which it comes. If the signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal named `MySignal` would have the following default names in the channel legend: `MySignal:1`, `MySignal:2`.

The Time Scope does not display signal names that were labeled within an unmasked subsystem. You must label all input signals to the Scope block that originate from an unmasked subsystem.

To change the appearance of any channel of any input signal in the Scope window, from the menu, select **View > Style**.

Show grid

When you select this check box, a grid appears in the active display of the Scope. To hide the grid, clear this check box. Tunable.

Plot signal(s) as magnitude and phase

When you select this check box, the Scope splits the active display into a magnitude plot and a phase plot. By default, this check box is cleared. If the input signal is complex valued, the Scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes within the same active display, as shown in the following figure.

Selecting this check box and clicking the **Apply** or **OK** button changes the active display. The magnitude of the input signal appears on the top axes and its phase, in degrees, appears on the bottom axes. See the following figure.

This feature is particularly useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box always produces the same behavior. The phase is 0 for nonnegative input and 180 degrees for negative input. Tunable.

Minimum Y-limit

Specify the minimum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a minimum value of –180 degrees.

Maximum Y-limit

Specify the maximum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a maximum value of 180 degrees.

Y-axis label

Specify the text for the scope to display to the left of the *y*-axis. Tunable.

This parameter becomes invisible when you select the **Plot signal(s) as magnitude and phase** check box. When you enable that parameter, the *y*-axis label always appears as Magnitude on the top axes and Phase on the bottom axes.

Tools — Plot Navigation Options

The Tools:Plot Navigation Options dialog box provides you with the ability to automatically zoom in on and zoom out of your data, and to scale the axes of the Time Scope. In the Time Scope menu, select **Tools > Axes Scaling Options** to open this dialog box.

Alternatively, you can open this dialog box from the Time Scope — Configuration dialog box. In the Time Scope menu, select **File > Configuration** to open the Time Scope — Configuration dialog box. Click the **Tools** pane, and the following dialog box appears.

Time Scope

The **Tools** pane in the Time Scope – Configuration dialog box contains the tools that appear on the Time Scope GUI. By default, the **Plot Navigation** tool is enabled. If you disable the tool by clearing the **Enabled** check box, the axes scaling and zoom tools no longer display in the Time Scope GUI. You can reenable the tool at any time by selecting the **Enabled** check box.

By default, the **Measurements** tool is also enabled. If you disable the tool by clearing the **Enabled** check box, the Time Domain Measurements tools no longer display in the Time Scope GUI. See the "Measurements Panels" on page 2-1456 section for more information. You can reenable the tool at any time by selecting the **Enabled** check box.

To open the Time Scope — Tools:Plot Navigation Options dialog box, select the **Plot Navigation** tool, and then click the **Options** button.

Parameters

The Tools:Plot Navigation Options dialog box appears as follows.

Axis Scaling

Specify when the Scope should automatically scale the axes. You can select one of the following options:

- Manual — When you select this option, the Scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:

    - Select **Tools > Scale axes limits**.

    - Press the **Scale Axes Limits** toolbar button.

    - When the scope GUI is the active window, press **Ctrl+A**.

- Auto — When you select this option, the Scope scales the axes as needed, both during and after simulation. Selecting this option enables the **Do not allow Y-axis limits to shrink** check box. By default, this parameter is selected, and the Scope does not shrink the *y*-axis limits when scaling the axes.

- Once at stop — Selecting this option causes the Scope to scale the axes each time simulation is stopped. The Scope does not scale the axes during simulation.

This parameter is Tunable.

Do not allow Y-axis limits to shrink

When you select this parameter, the *y*-axis limits are only allowed to grow during axes scaling operations. If you clear this check box, the *y*-axis limits may shrink during axes scaling operations.

This parameter appears only when you select Auto for the **Axis Scaling** parameter. When you set the **Axis Scaling** parameter to Manual or Once at stop, the *y*-axis limits are allowed to shrink. Tunable.

Y-axis Data range (%)

Set the percentage of the *y*-axis that the Scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to 100, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this parameter to 30, the Scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable.

Y-axis Align

Specify where the Scope should align your data with respect to the *y*-axis when it scales the axes. You can select Top, Center, or Bottom. Tunable.

Scale X-axis limits

Check this box to allow the Scope to scale the *x*-axis limits when it scales the axes. Tunable.

X-axis Data range (%)

Set the percentage of the *x*-axis that the Scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to 100, the Scope scales the *x*-axis limits such that your data uses the entire*x*-axis range. If you then set this parameter to 30, the Scope increases the *x*-axis range such that your data uses only 30% of the *x*-axis range. Use the *x*-axis **Align** parameter to specify data placement with respect to the *x*-axis.

This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

X-axis Align

Specify how the Scope should align your data with respect to the *x*-axis: Left, Center, or Right. This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

Sources — Simulink Options

The Sources:Simulink Options dialog box lets you control the number of samples for each input signal that Time Scope holds in its memory cache. In the Time Scope menu, select **View > Data History Options** to open this dialog box.

Data History Options
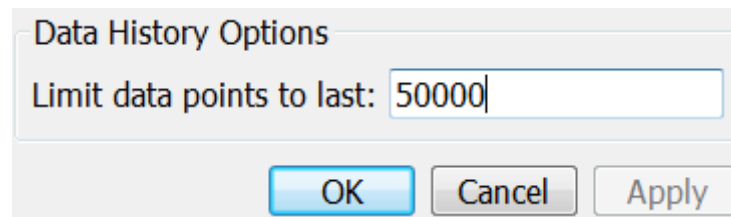
Limit data points to last

Specify the number of data points for each input signal that Time Scope holds in its memory cache. The number of data points is equal to the product of the number of channels in the signal and the number of time samples. The default setting is 50000.

Examples

The first few examples illustrate how to use the Time Scope block to view a variety of input signals in the time domain.

- "Example: Display Complex-valued Input Signal" on page 2-1512
- "Example: Display Variable-size Input Signal" on page 2-1514
- "Example: Display Simulink Enumeration Input Signal" on page 2-1516

The remaining examples demonstrate how to use the Measurements Panels in the Time Scope GUI to glean information about the input signals.

Time Scope

Example: Display Complex-valued Input Signal

At the MATLAB command prompt, type ex_timescope_complexinp to open the example model. The following Simulink model appears.
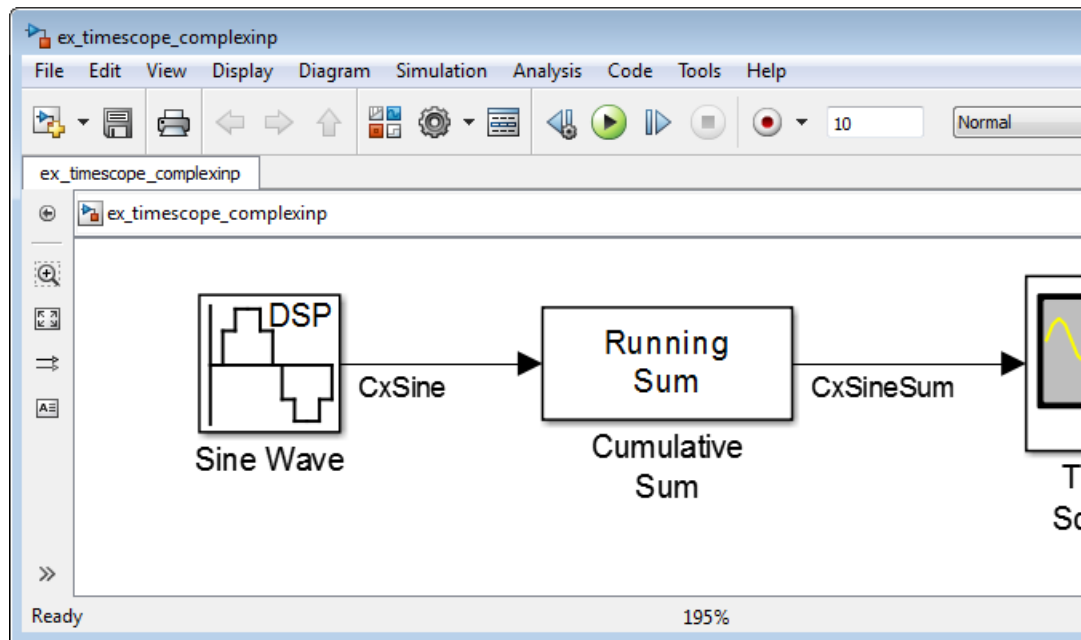
By default, when the input is a complex-valued signal, Time Scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes within the same active display. Run your model to see the time domain output, as shown in the following figure.
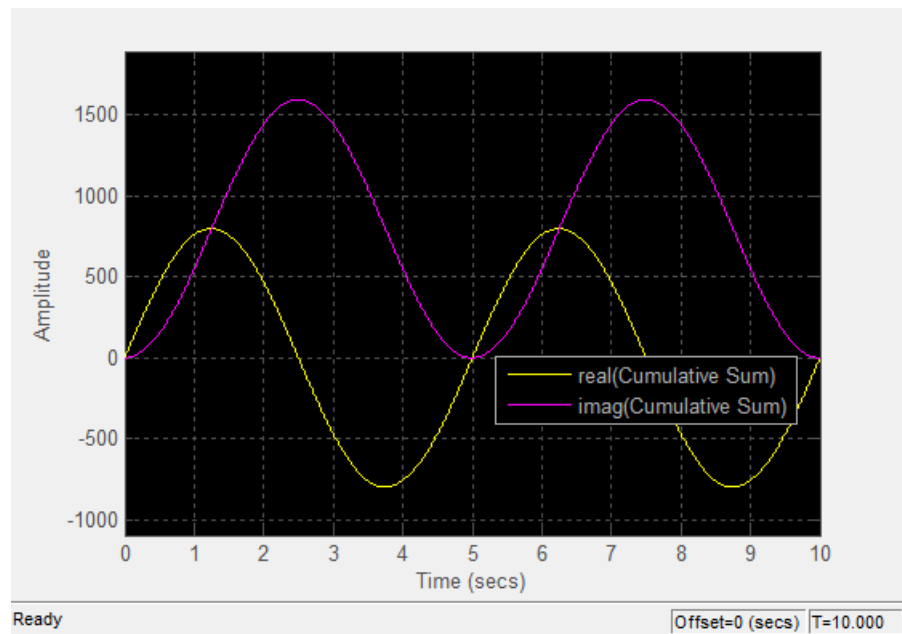
The Properties dialog box controls the visual configuration settings of the Time Scope displays. From the Time Scope menu, select **View > Properties** to open this dialog box. Selecting the **Plot signal(s) as magnitude and phase** check box specifies for Time Scope to plot the magnitude and phase of the input signal. The magnitude and phase appear on two separate axes within the same active display. After you select this check box, click **OK**. The active display shows the magnitude of the input signal on the top axes. The signal phase, in degrees, appears on the bottom axes. See the following figure.

Example: Display Variable-size Input Signal

At the MATLAB command prompt, type `ex_timescope_varsize` to open the example model. The following Simulink model appears.
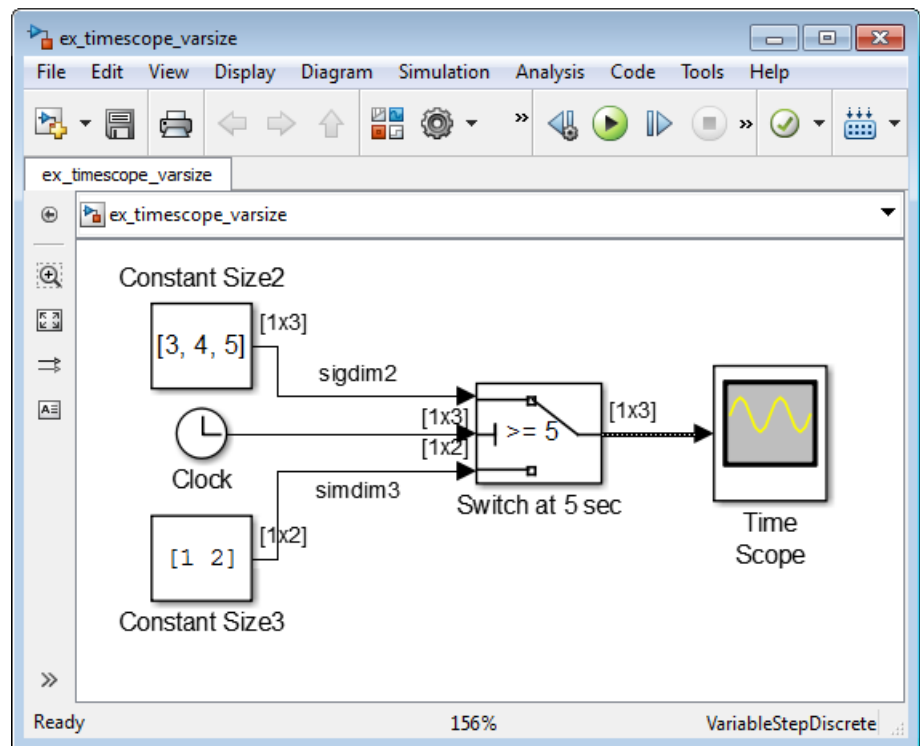
In this example, the size of the input signal to the Time Scope block changes as the simulation progresses. When the simulation time is less than 5 seconds, Time Scope plots the signal connected to the third input port of the Switch block, which has signal dimensions 1 by 2. After 5 seconds, Time Scope plots the signal connected to the first input port of the Switch block, which has signal dimensions 1 by 3. Run your model to see the time domain output, as shown in the following figure.

Time Scope
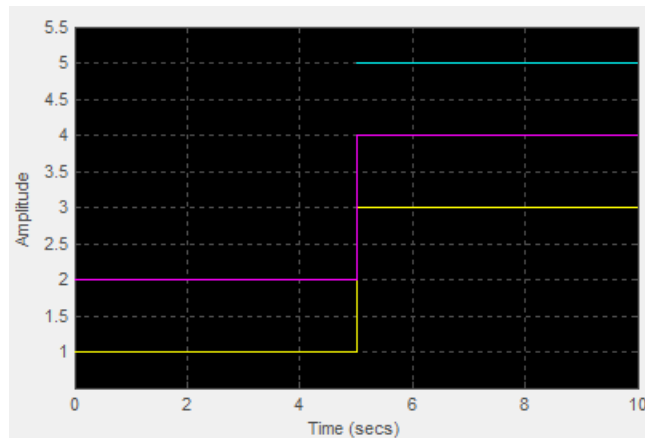
As you can see in the figure, the third line on the display, colored cyan, appears only after 5 seconds.

Example: Display Simulink Enumeration Input Signal

At the MATLAB command prompt, type `ex_timescope_slenum` to open the example model. The following Simulink model appears.
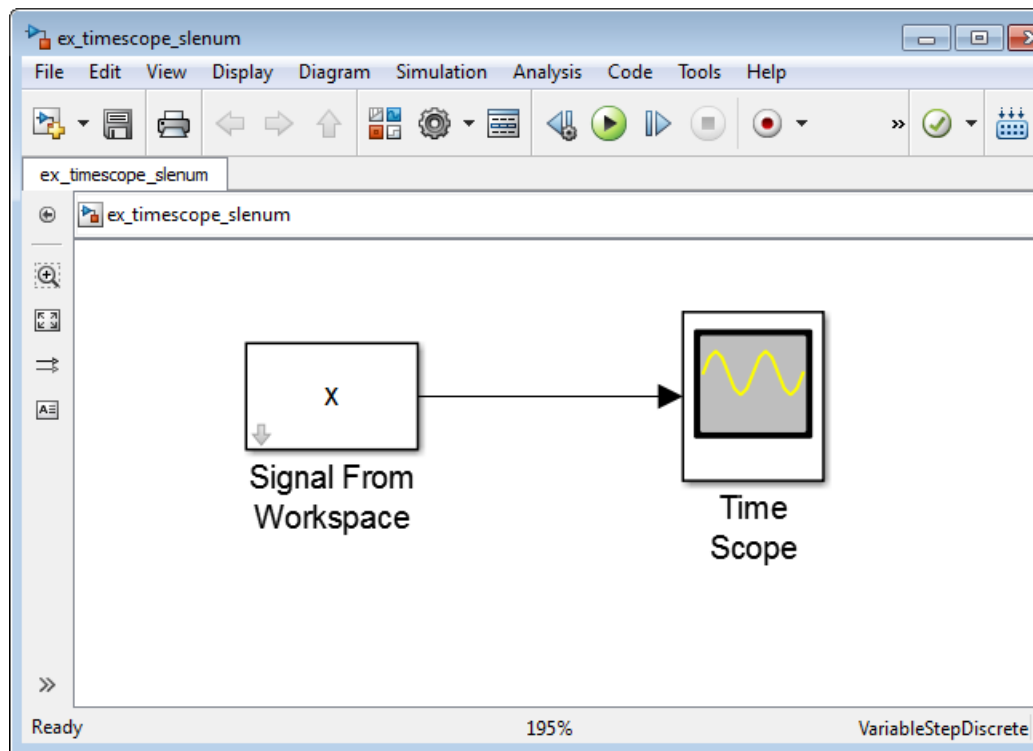
In this example, Simulink imports the variable *x*, which is a Simulink enumeration data type, from the MATLAB workspace. This variable is created when the model loads because the commands that construct it reside in the Preload function. To view these commands, in the Simulink menu, select **File > Model Properties > Model Properties**. The following lines of MATLAB code appear.

```
if ~exist('BasicColors','class')
    Simulink.defineIntEnumType('BasicColors', ...
        {'Red', 'Yellow', 'Blue'}, ...
        [0;1;2], ...
        'Description', 'Basic colors', ...
        'DefaultValue', 'Blue', ...
```

```
            'AddClassNameToEnumNames', true);
end
x = [BasicColors(0), BasicColors(2), BasicColors(1)];
```

The Signal from Workspace block has a Sample time of 3 seconds. Thus, the input signal changes to the next value in the vector *x* every 3 seconds. Run your model to see the time domain output, as shown in the following figure.
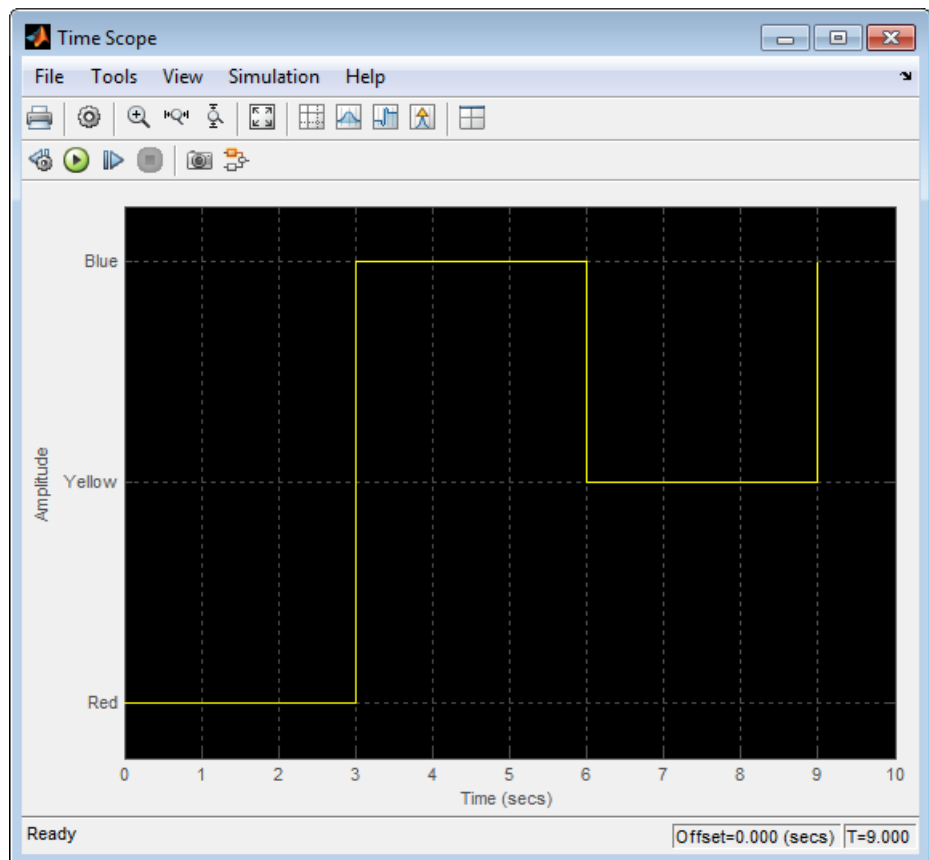
As you can see in the figure, the *y*-axis shows the units of amplitude as one of Red, Yellow, or Blue. The input signal value changes from Red to Blue at 3 seconds and from Blue to Yellow at 6 seconds.

Example: Use Bilevel Measurements Panel with Clock Input Signal

At the MATLAB command prompt, type `ex_timescope_clockex` to open the example model. The following Simulink model appears.
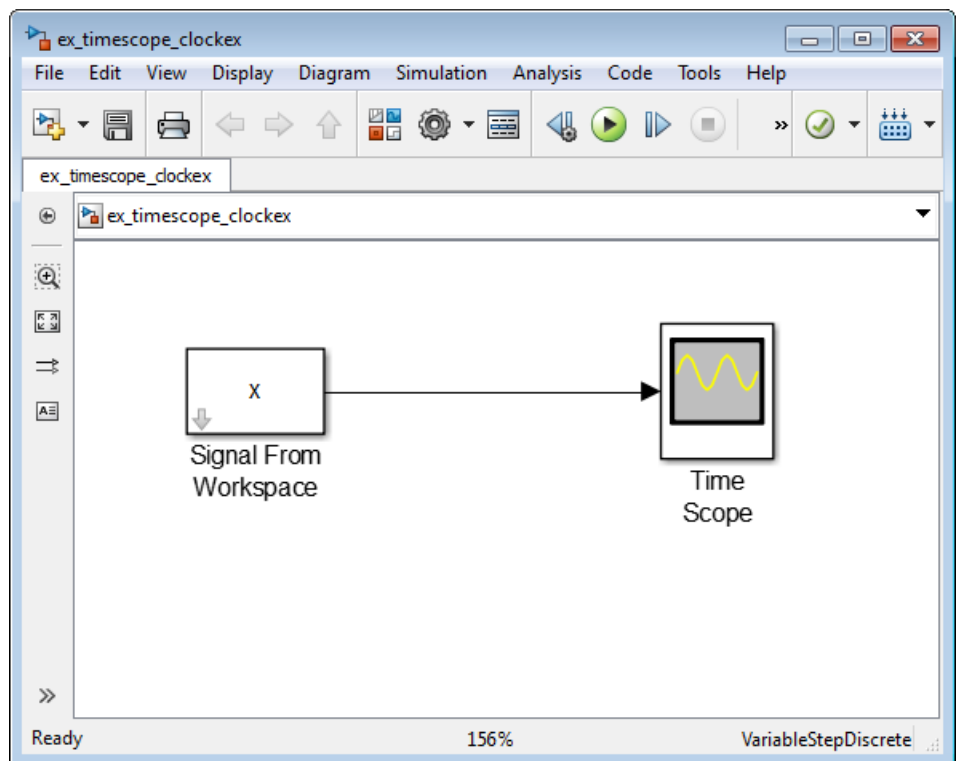
In this example, Simulink imports the variable *x* from the MATLAB workspace. This variable is created when the model loads because the commands that construct it reside in the Preload function. To

view these commands, in the Simulink menu, select **File > Model Properties > Model Properties**. The Model Properties dialog box appears. Click the **Callbacks** tab. The following lines of MATLAB code appear.

```
load clockex;
ts = t(2)-t(1);
```

Run your model to see the time domain output. To show the **Bilevel Measurements** panel, in the Time Scope menu, select **Tools > Measurements > Bilevel Measurements**. To collapse the **Transitions** pane, click the pane collapse button (▼) next to that label. To expand the **Settings** pane and the **Overshoots / Undershoots** pane, click the pane expand button (▶) next to each label. The Time Scope figure appears as shown in the following figure.

As you can see in the figure, the value for the rising edge **Settling Time** parameter is initially not displayed. The reason for this is that the default value for the **Settle Seek** parameter is too large for this example. In this case, the settle seek time is longer than the entire simulation duration. Enter a value for settle seek of `2e-6`, and press the **Enter** key. Time Scope now displays a rising edge settling time value of `118.392 ns`.

The settling time value displayed is actually the statistical average of the settling times for all five rising edges. To show the settling time for only one rising edge, you can zoom in on that transition. In the Time Scope toolbar, click the Zoom X button ( ). Click on the display near a value of 2 microseconds on the *time*-axis. Drag to the right and release near a value of 4 microseconds on the *time*-axis. Time Scope updates the rising edge **Settling Time** value to reflect the new time window, as shown in the following figure.
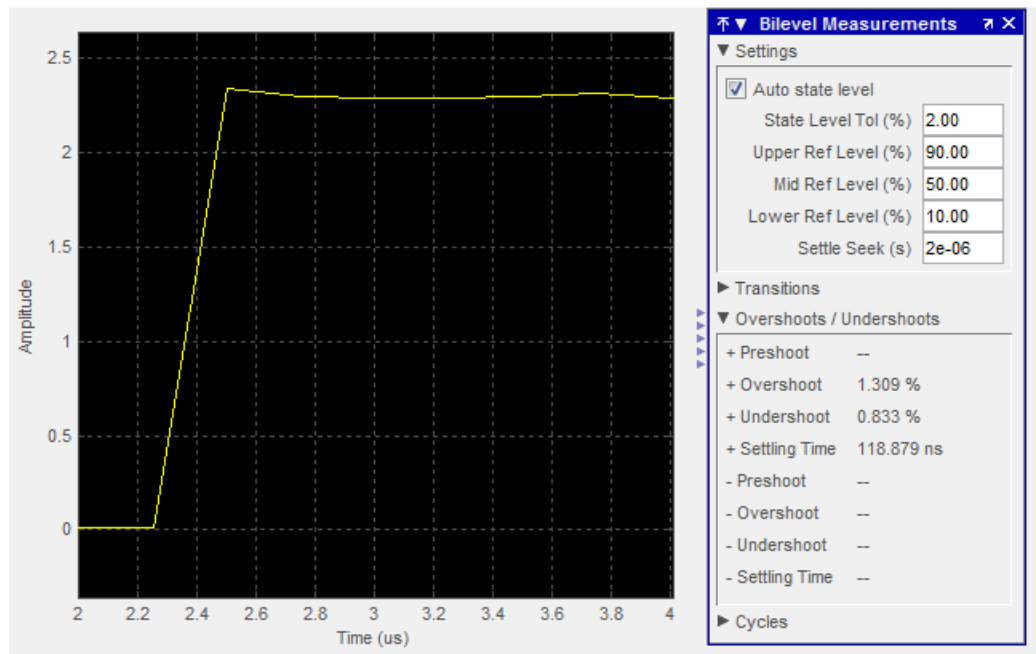
Example: Find Heart Rate Using Peak Finder Panel with ECG Input Signal

At the MATLAB command prompt, type ex_timescope_heartbeat to open the example model. The following Simulink model appears.
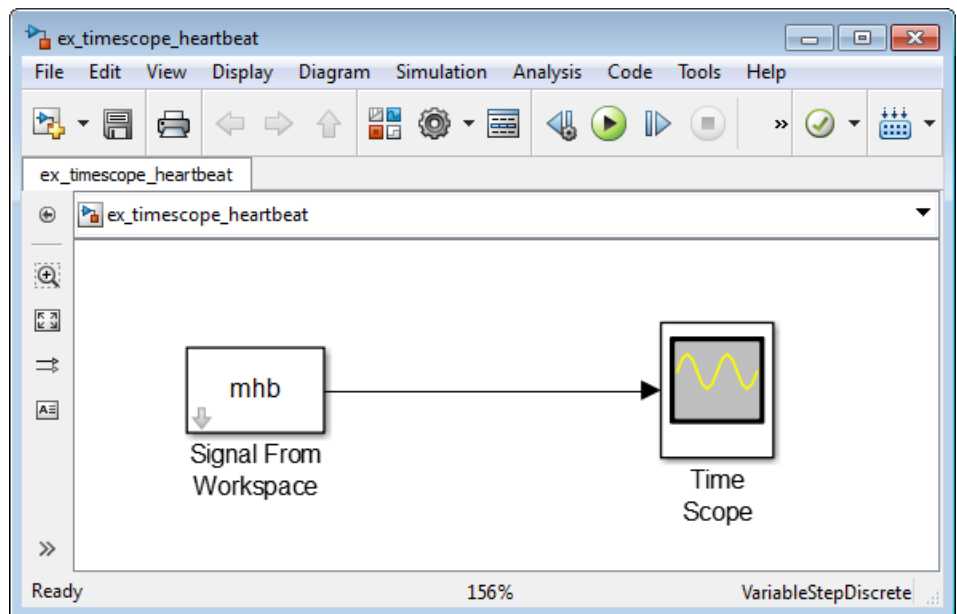
In this example, Simulink imports the variable *mhb* from the MATLAB workspace. The variable *mhb* is created when the model loads because the commands that construct *mhb* are in the Preload function. To view these commands, in the Simulink menu, select **File > Model Properties > Model Properties**. The Model Properties dialog box appears. Click the **Callbacks** tab. The following lines of MATLAB code appear.

```
x1 = 3.5*ecg(2700).';
y1 = sgolayfilt(kron(ones(1,13),x1),0,21);
n = (1:30000)';
del = round(2700*rand(1));
mhb = y1(n + del);
ts = 0.00025;
```

Time Scope

This example uses the Savitzky-Golay filter in Signal Processing Toolbox. For more information, see the sgolayfilt function reference page or run the `sgolaydemo` example.

Run your model to see the time domain output. To show the **Peak Finder** panel, in the Time Scope menu, select **Tools > Measurements > Peak Finder**. To expand the **Settings** pane, click the pane expand button (▶) next to that label. Enter a value for **Max Num of Peaks** of 10 and press the **Enter** key. Time Scope now displays in the **Peaks** pane a list of 10 peak amplitude values, and the times at which they occur, as shown in the following figure.
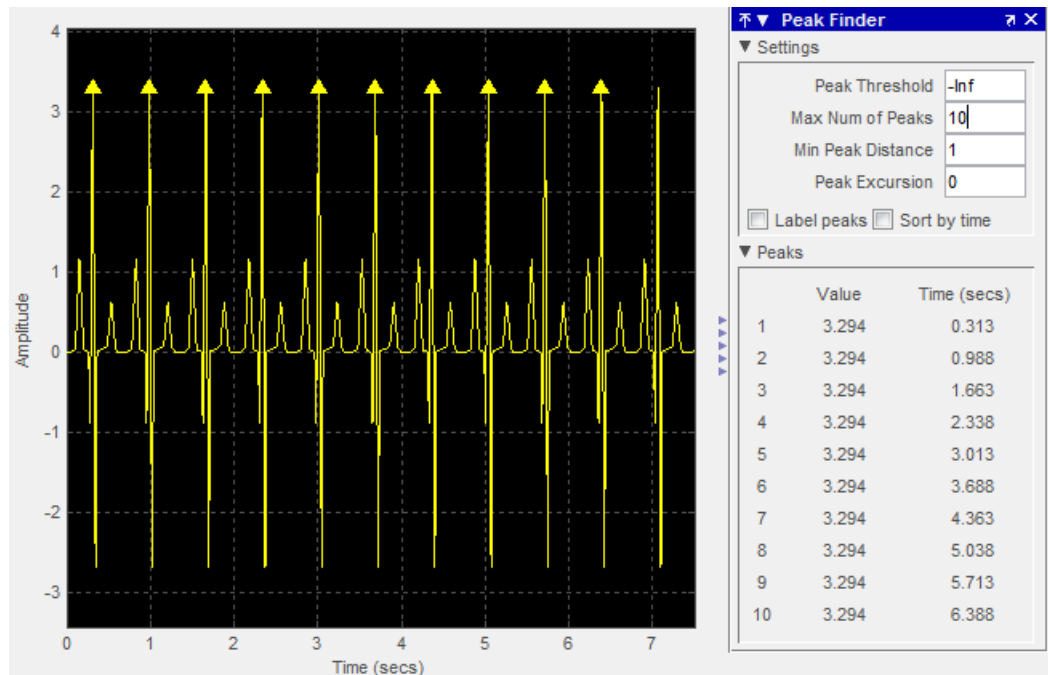
As you can see from the list of peak values, there is a constant time difference of 0.675 seconds between each heartbeat. Therefore, the heart rate of the ECG signal is given by the following equation.

$$\frac{60\dfrac{\text{sec}}{\text{min}}}{0.675\dfrac{\text{sec}}{\text{beat}}} = 88.89\frac{\text{beats}}{\text{min}}(\text{bpm})$$

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	• Simulink enumerations

Supported Simulation Modes

You can use the Time Scope block in models running the following supported simulation modes.

Mode	Supported	Notes and Limitations
Normal	Yes	
Accelerator	Yes	

Mode	Supported	Notes and Limitations
Rapid Accelerator	Yes	You can use Rapid Accelerator mode as a method to increase the execution speed of your Simulink model. Rapid Accelerator mode creates an executable that includes the solver and model methods. This executable resides outside MATLAB and Simulink. Rapid Accelerator mode uses External mode to communicate with Simulink. For more information about Rapid Accelerator mode, see "Acceleration" in the Simulink documentation.
PIL	No	
SIL	No	
External	Yes	You can use External mode to tune block parameters in real time and view block outputs in many types of blocks and subsystems. External mode establishes communication between a host system, where the Simulink environment resides, and a target system, where the executable runs after it is generated by the code generation and build process. For more information about External mode, see "Host/Target Communication" in the Simulink Coder documentation. Time Scope does not support data archiving. See "Set External Mode Data Archiving Parameters" in the Real-Time Windows Target documentation. Do not interactively open or close the Time Scope between the Connect and Start of the target. Attempting to do causes a checksum mismatch between the built model and the

Mode	Supported	Notes and Limitations
		model in session. Open the Time Scope either before building or after starting the simulation.

For more information about these modes, see "How Acceleration Modes Work" in the Simulink documentation.

See Also   Scope | dsp.TimeScope | Spectrum Scope | Vector Scope | sptool

How To   • Display Time-Domain Data in the Time Scope

Time-Varying Direct-Form II Transpose Filter (Obsolete)

Purpose   Apply variable IIR filter to input

Library   dspobslib

Description

Note This block is now just an implementation of the Digital Filter block.

Purpose   Apply variable lattice filter to input

Library   dspobslib

Description

Note This block is now just an implementation of the Digital Filter block.

Toeplitz

Purpose   Generate matrix with Toeplitz symmetry

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3

Description   The Toeplitz block generates a Toeplitz matrix from inputs defining the first column and first row. The top input (Col) is a vector containing the values to be placed in the first *column* of the matrix, and the bottom input (Row) is a vector containing the values to be placed in the first *row* of the matrix.

```
y = toeplitz(Col,Row)      % Equivalent MATLAB code
```

The other elements of the matrix obey the relationship

```
y(i,j) = y(i-1,j-1)
```

and the output has dimension [length(Col) length(Row)]. The $y(1,1)$ element is inherited from the Col input. For example, the following inputs

```
Col = [1 2 3 4 5]
Row = [7 7 3 3 2 1 3]
```

produce the Toeplitz matrix

$$
\begin{bmatrix}
1 & 7 & 3 & 3 & 2 & 1 & 3 \\
2 & 1 & 7 & 3 & 3 & 2 & 1 \\
3 & 2 & 1 & 7 & 3 & 3 & 2 \\
4 & 3 & 2 & 1 & 7 & 3 & 3 \\
5 & 4 & 3 & 2 & 1 & 7 & 3
\end{bmatrix}
$$

When you select the **Symmetric** check box, the block generates a symmetric (Hermitian) Toeplitz matrix from a single input, u, defining both the first row and first column of the matrix.

```
y = toeplitz(u)        % Equivalent MATLAB code
```

The output has dimension [length(u) length(u)]. For example, the Toeplitz matrix generated from the input vector [1 2 3 4] is

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

The Toeplitz block supports real and complex floating-point and fixed-point inputs.
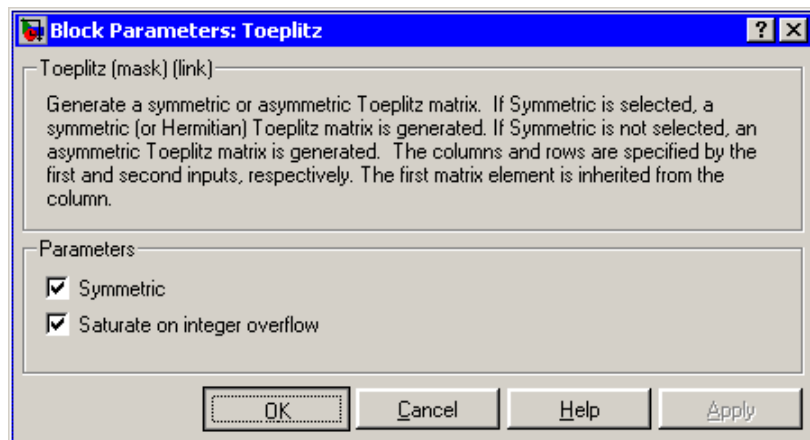
Dialog Box

Symmetric

When selected, enables the single-input configuration for symmetric Toeplitz matrix output.

Saturate on integer overflow

When you generate a symmetric Toeplitz matrix with this block, if the input vector is complex, the output is a symmetric Hermitian matrix whose elements satisfy the relationship

Toeplitz

$$y(i, j) = \text{conj}(y(j, i))$$

For fixed-point signals the conjugate operation could result in an overflow. When you select this parameter, overflows saturate. This parameter is only visible with the **Symmetric** parameter is selected. This parameter is ignored for floating-point signals.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers (real signals only)
Toep Col	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Toep Row	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Constant Diagonal Matrix	DSP System Toolbox
toeplitz	MATLAB

To Audio Device

| **Purpose** | Write audio data to computer's audio device |

| **Library** | Sinks |

dspsnks4

Description
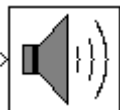
The To Audio Device block sends audio data to your computer's audio device. This block is not supported for use with the Simulink Model block.

Use the **Device** parameter to specify the device to which you want to send the audio data. This parameter is automatically populated based on the audio devices installed on your system. If you plug or unplug an audio device from your system, type `clear mex` at the MATLAB command prompt to update the list.

Select the **Inherit sample rate from input** check box if you want the block to inherit the sample rate of the audio signal from the input to the block. If you clear this check box, the **Sample rate (Hz)** parameter appears on the block. Use this parameter to specify the number of samples per second in the signal.

The range of supported audio device sample rates and data type formats, depend on both the sound card and the API which is chosen for the sound card.

Use the **Device data type** to specify the data type of the audio data that is sent to the device. You can choose:

- `8-bit integer`
- `16-bit integer`
- `24-bit integer`
- `32-bit float`
- `Determine from input data type`

If you choose `Determine from input data type`, the following table summarizes the block's behavior.

Input Data Type	Device Data Type
Double-precision floating point or single-precision floating point	32-bit floating point
32-bit integer	24-bit integer
16-bit integer	16-bit integer
8-bit integer	8-bit integer

If you choose Determine from input data type and the device does not support the input data type, the block uses the next lowest-precision data type supported by the device.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra .dll files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See packNGo for more information.

Buffering

The To Audio Device block buffers the data from a Simulink signal using the process illustrated by the following figure.
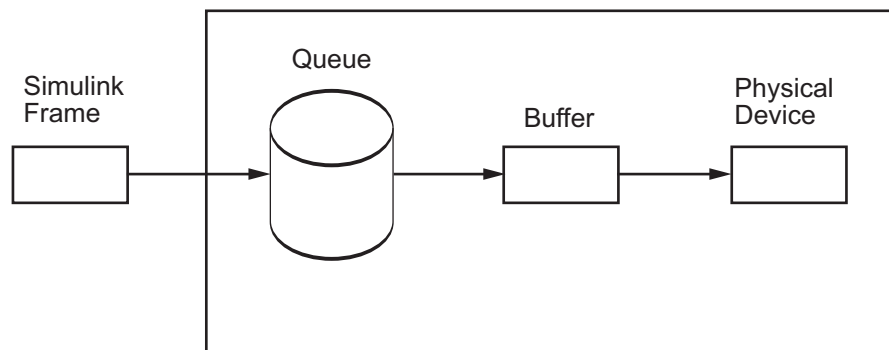
To Audio Device Block

1 At the start of the simulation, the queue is filled with silence. Specify the size of this queue using the **Queue duration (seconds)** parameter. As Simulink runs, the block appends Simulink frames to the bottom of the queue.

2 At each time step, the blocks sends a buffer of samples from the top of the queue to the audio device. Select the **Automatically determine buffer size** check box to allow the block to use a conservative buffer size. See the From Audio Device block reference page for the equation the block uses to calculate this buffer size. If you clear this check box, the **Buffer size (samples)** parameter appears on the block. Use this parameter to specify the size of the buffer in samples.

3 The block writes the buffer of audio data to the device. If the queue did not contain enough data to completely fill the buffer, the block fills the remaining portion of the buffer with zeros. This data has a the data type specified by the **Device data type** parameter.

When the simulation throughput rate is lower than the hardware throughput rate, the queue, which is initially full, becomes empty. If the queue is empty, the block sends zeros (silence) to the audio device. When the simulation throughput rate is higher than the hardware throughput rate, the To Audio Device block waits to write data to the queue.

To minimize the chance of dropouts, the block checks to make sure the queue duration is at least as large as the maximum of the buffer size and the frame size. If it is not, the queue duration is automatically set to this maximum value.

Troubleshooting

When Simulink cannot keep up with an audio device that is operating in real time, the queue becomes empty and gaps occur in the audio data that the block sends to the device. Here are several ways to deal with this situation:

- *Increase the queue duration.*

The **Queue duration (seconds)** parameter specifies the duration of the signal, in seconds, that can be buffered during the simulation. This is the maximum length of time that the block's data supply can lag the hardware's data demand.

- *Increase the buffer size.*

  The size of the buffer processed in each interrupt from the audio device affects the performance of your model. If the buffer is too small, a large portion of hardware resources are used to write data to the device. If the buffer is too big, Simulink must wait for the device to empty the buffer before it can write the data to the queue, which introduces 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 use frame-based processing throughout the model to reduce the amount of block-to-block communication overhead. This can 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 Simulink Coder code generation software. Native code runs much faster than Simulink and should provide rates adequate for real-time audio processing.

Other ways to improve throughput rates include simplifying the model and running the simulation on a faster PC processor. For other ideas on improving simulation performance, see "Delay and Latency" and "Performance" in the Simulink documentation.

Channel-to-Speaker Mapping on Windows Operating Systems

The To Audio Device and From Audio Device blocks can support multiple channels. On Windows operating systems, the channel-to-speaker mapping is defined as listed below. This mapping only applies when

your sound card is properly configured and capable of receiving the audio data you send. If the number of channels on the card does not match the number of channels on the block, or if you specify a data type for the **Device data type** parameter that is not supported by your device, the Windows mixer intervenes to translate from one format to another. If the Windows mixer does intervene, the channel-to-speaker mapping might differ from what is specified here.

- Single channel input — Front center speaker

  On systems with two speakers, the front center channel is split between the right and left speakers.

- Multichannel input — Channels are assigned to speakers as follows:

  - One channel — Front center

  - Two channels — Front left, front right

  - Four channels — Front left, front right, rear left, rear right

  - Six channels — Front left, front right, front center, low frequency, rear left, rear right

  - Eight channels — Front left, front right, front center, low frequency, rear left, rear right, front left center, front right center

  - For all other channel combinations, the channel assignment is dictated by the audio card.

Audio Hardware API

The To Audio Device and From Audio Device blocks use the open-source PortAudio library in order to communicate with the audio hardware on a given computer. The PortAudio library supports a range of API's designed to communicate with the audio hardware on a given platform. The following API choices were made when building the PortAudio library for the DSP System Toolbox product:

- Windows: DirectSound, WDM-KS, ASIO
- Linux: OSS, ALSA

- Mac: CoreAudio

To select or change the Audio Hardware API, select **Preferences** from the MATLAB Toolstrip. Then select DSP System Toolbox from the tree menu. In the **DSP System Toolbox Preferences** dialog box, the option is disabled if no device corresponds to that particular audio API.

Example
See the Positional Audio example for a demonstration of how to use this block. You can open this example by typing dspAudioPos at the MATLAB command line.
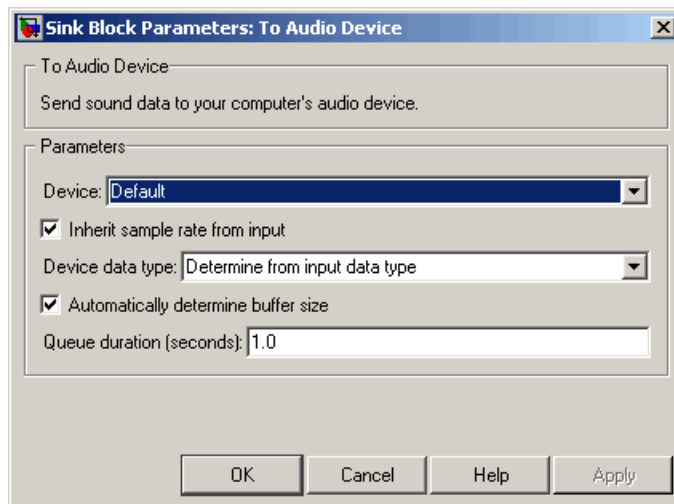
Dialog Box

Device
Specify which device to send the audio data to.

Inherit sample rate from input
Select this check box if you want the block to inherit the sample rate of the audio signal from the input to the block.

Sample rate (Hz)
Specify the number of samples per second in the signal. This parameter is visible when the **Inherit sample rate from input** check box is cleared.

Device data type
Specify the data type of the audio data sent to the device.

Automatically determine buffer size
Select this check box to allow the block to calculate a conservative buffer size.

Buffer size (samples)
Specify the size of the buffer. This parameter is visible when the **Automatically determine buffer size** check box is cleared.

Queue duration (seconds)
Specify the size of the queue in seconds.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• 32-bit signed integers
	• 16-bit signed integers
	• 8-bit unsigned integers

See Also

From Audio Device	DSP System Toolbox
To Multimedia File	DSP System Toolbox
audioplayer	MATLAB
sound	MATLAB
dsp.AudioPlayer	System Object

Purpose   Write video frames and audio samples to multimedia file

Library   Sinks

dspsnks4

Description
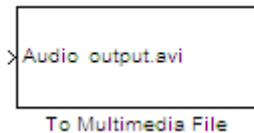
To Multimedia File

The To Multimedia File block writes video frames, audio samples, or both to a multimedia (.avi, .wav, .wma, or .wmv) file.

You can compress the video frames or audio samples by selecting a compression algorithm. You can connect as many of the input ports as you want. Therefore, you can control the type of video and/or audio the multimedia file receives.

Note This block supports code generation for platforms that have file I/O available. You cannot use this block with Real-Time Windows Target software, because that product does not support file I/O.

This block performs best on platforms with Version 11 or later of Windows Media® Player software. This block supports only uncompressed RGB24 AVI files on Linux® and Mac platforms.

Windows 7 UAC (User Account Control), may require administrative privileges to encode WMV and WMA files.

The generated code for this block relies on prebuilt library files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra library files when doing so. The

packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

To run an executable file that was generated from a model containing this block, you may need to add precompiled shared library files to your system path. See "Understanding Code Generation" for details.

Cross-Platform Supported File Formats for Audio Files

Audio files can be of the following formats on all platforms:

- WAV
- FLAC
- OGG

Ports

Port	Description
Image	M-by-N-by-3 matrix RGB, Intensity, or YCbCr 4:2:2 signal.
R, G, B	Matrix that represents one plane of the RGB video stream. Inputs to the R, G, or B port must have the same dimensions and data type.
Audio	Vector of audio data
Y, Cb, Cr	Matrix that represents one frame of the YCbCr video stream. The Y, Cb, and Cr ports use the following dimensions: Y: $M \times N$ Cb: $M \times \dfrac{N}{2}$ Cr: $M \times \dfrac{N}{2}$

Dialog Box

The **Main** pane of the To Multimedia File block dialog appears as follows.
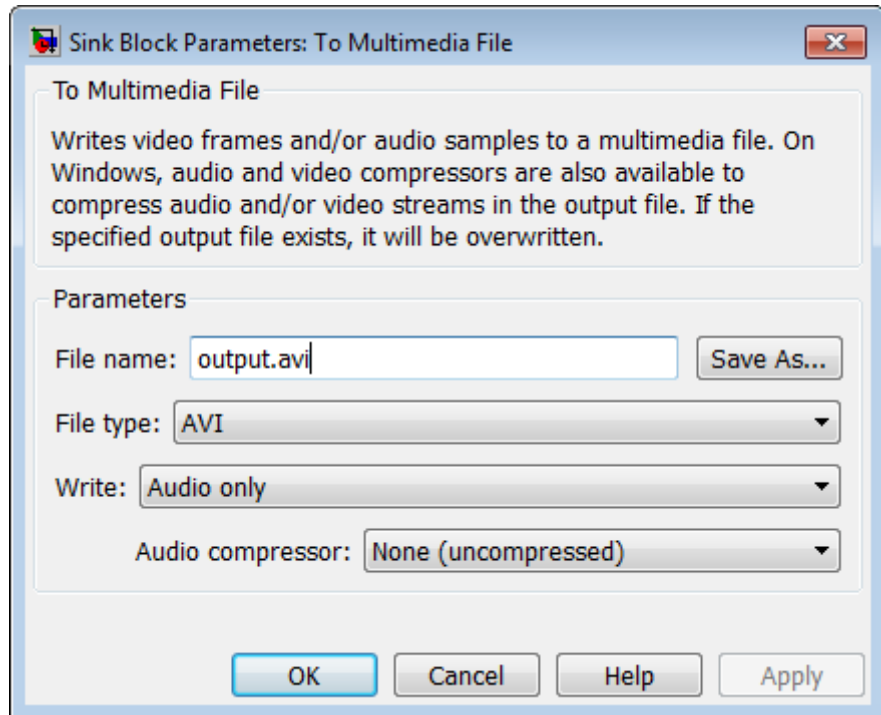
File name
> Specify the name of the multimedia file. The block saves the file in your current folder. To specify a different file or location, click the **Save As...** button.

File type
> Specify the file type of the multimedia file. You can select `avi`, `wav`, `wma`, or `wmv`.

Write

Specify whether the block writes video frames, audio samples, or both to the multimedia file. You can select Video and audio, Video only, or Audio only.

Audio compressor

Select the type of compression algorithm to use to compress the audio data. This compression reduces the size of the multimedia file. Choose None (uncompressed) to save uncompressed audio data to the multimedia file.

Note The other items available in this parameter list are the audio compression algorithms installed on your system. For information about a specific audio compressor, see the documentation for that compressor.

Audio data type

Select the audio data type. You can use the **Audio data type** parameter only for uncompressed wave files.

Video compressor

Select the type of compression algorithm to use to compress the video data. This compression reduces the size of the multimedia file. Choose None (uncompressed) to save uncompressed video data to the multimedia file.

Note The other items available in this parameter list are the video compression algorithms installed on your system. For information about a specific video compressor, see the documentation for that compressor.

File color format

Select the color format of the data stored in the file. You can select either RGB or YCbCr 4:2:2.

Image signal

Specify how the block accepts a color video signal. If you select One multidimensional signal, the block accepts an *M*-by-*N*-by-*P* color video signal, where *P* is the number of color planes, at one port. If you select Separate color signals, additional ports appear on the block. Each port accepts one *M*-by-*N* plane of an RGB video stream.

Supported Data Types

For the block to display video data properly, double- and single-precision floating-point pixel values must be between 0 and 1. Any other data type requires the pixel values between the minimum and maximum values supported by their data type.

Check the specific codecs you are using for supported audio rates.

Port	Supported Data Types	Supports Complex Values?
Image	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16- 32-bit signed integers • 8-, 16- 32-bit unsigned integers	No
R, G, B	Same as Image port	No
Audio	• Double-precision floating point • Single-precision floating point • 16-bit signed integers • 32-bit signed integers • 8-bit unsigned integers	No
Y, Cb, Cr	Same as Image port	No

To Multimedia File

Purpose   Send audio data to standard Windows audio device in real time

Library   dspwin32

Description
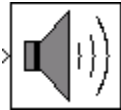
> **Note** The To Wave Device block is still supported but is likely to be
> obsoleted in a future release. We strongly recommend replacing this
> block with the To Audio Device block.

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. The data is sent to the
hardware in uncompressed pulse code modulation (PCM) format, and
should typically be sampled at one of the standard Windows audio
device rates: 8000, 11025, 22050, or 44100 Hz. Some hardware might
support other rates in addition to these.

> **Note** Models that contain both the To Wave Device block and the From
> Wave Device block require a duplex-capable sound card.

The **Use default audio device** check box allows the To Wave Device
block to detect and use the system's default audio hardware. You should
select this option for systems that have a single sound device installed,
or when the default sound device on a multiple-device system is your
desired target. When the default sound device is *not* your desired
output device, clear **Use default audio device**, and set the desired
hardware in the **Audio device** parameter. This parameter lists the
names of the installed audio devices.

The block input can contain audio data from a mono or stereo signal.
A mono signal is represented as either a sample-based scalar or a
frame-based length-M vector, where M is frame size. A stereo signal
is represented as a sample-based length-2 vector or a frame-based
M-by-2 matrix.

To Wave Device (Obsolete)

When the input data type is `uint8`, the block conveys the signal samples to the audio device using 8 bits. When the input data type is `double`, `single`, `int16`, or fixed point with a word length of 16 and a fraction length of 15, the block conveys the signal samples to the audio device using 16 bits by default. For inputs of data type `double` and `single`, you can also set the block to convey the signal samples using 24 bits by selecting the **Enable 24-bit output for double- and single-precision input signals** check box. The 24-bit sample width requires more memory but in general yields better fidelity.

The amplitude of the input must be in a valid range that depends on the input data type, as shown in the following table. Amplitudes outside the valid range are clipped to the nearest allowable value.

Input Data Type	Valid Input Amplitude Range
`double`	$-1 \leq amplitude < 1$
`single`	$-1 \leq amplitude < 1$
`int16`	$-32768 \leq amplitude \leq 32767$
`uint8`	$0 \leq amplitude \leq 255$
Fixed point with a word length of 16 and a fraction length of 15	$-1 \leq amplitude \leq 1 - 2^{-15}$

Buffering

Because audio devices generate real-time audio output, the Simulink environment must maintain a continuous flow of data to a device throughout 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 within Simulink rather than as generated code. Simulink execution speed can vary 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 available to the hardware on demand.

Note This block requires real-time execution of the parent model for best performance.

The following block parameters control the memory management for this block:

- **Queue duration**

- **Automatically determine internal buffer size** or **User-defined internal buffer size**

- **Initial output delay**
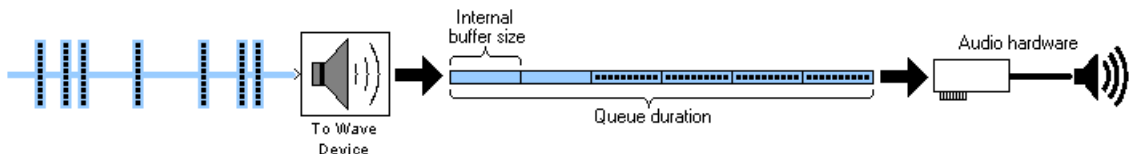
The **Queue duration** parameter defines the overall size of the block's buffer. The block reads in chunks of data in the size of the input dimensions and stores them in the buffer. The internal buffer size defines the dimensions of the block output to the hardware. You can define the internal buffer size yourself in the **User-defined internal buffer size parameter**. If you select **Automatically determine internal buffer size** instead, the internal buffer size is calculated for you according to the following rules:

To Wave Device (Obsolete)

- If the input to the block has a frame size of 32 samples or larger, the internal buffer size be the same as the input frame size.

- If the input to the block has a frame size smaller than 32 samples, the internal buffer size is based on the input sample rate according to the following table, where

$$F_s = sampling\ frequency = \frac{1}{sample\ time}$$

F_s (Hz)	Internal Buffer Size (samples)
$F_s < 8000$	$\min(64, 2 * F_s)$
$8000 \leq F_s < 22{,}050$	128
$22{,}050 \leq F_s < 44{,}100$	256
$44{,}100 \leq F_s < 96{,}000$	512
$F_s \geq 96{,}000$	1024

To minimize the chance of dropouts, the block checks to make sure that the queue duration is at least as big as twice the internal buffer size. If it is not, the queue duration is automatically set to twice the internal buffer size.

The **Initial output delay** parameter enables you to preload the buffer before the block starts to output data to the audio device, which can be helpful for models that do not run in real time. However, for real-time applications, it is best to set the initial output delay to zero (one frame of delay), or as close to zero as possible.

Troubleshooting

If you are getting undesirable audio output using the To Wave Device block, first determine whether your model can run in real time. Replace the To Wave Device block with a To Wave File block, run the model, and compare the model's simulation stop time to the elapsed time on your watch. If the model simulation stop time is less than the elapsed time on your watch, your model can probably run in real time. Then,

- If your model can run in real time,

  1 Select **Automatically determine internal buffer size**. This alone might solve the problem. If not,

  2 Try increasing the **Queue duration** parameter to a relatively large value, such as 0.5 s.

If one or both of these options restores desirable audio output, you can try reducing the internal buffer size and/or queue duration until the quality of the audio output again degrades.

- If your model is not running in real time, try to make it run in real time by

  1 Optimizing the model (using a more efficient implementation), or

  2 Using a Simulink "Acceleration" mode, or

  3 Generating stand-alone code

  If none of these are possible, but the model only runs for a short period of time, set the **Queue duration** parameter to a size equal to a significant fraction of the model stop time and use a similarly large initial delay. This is not an optimal solution, but might work in some cases.

**Dialog
Box**
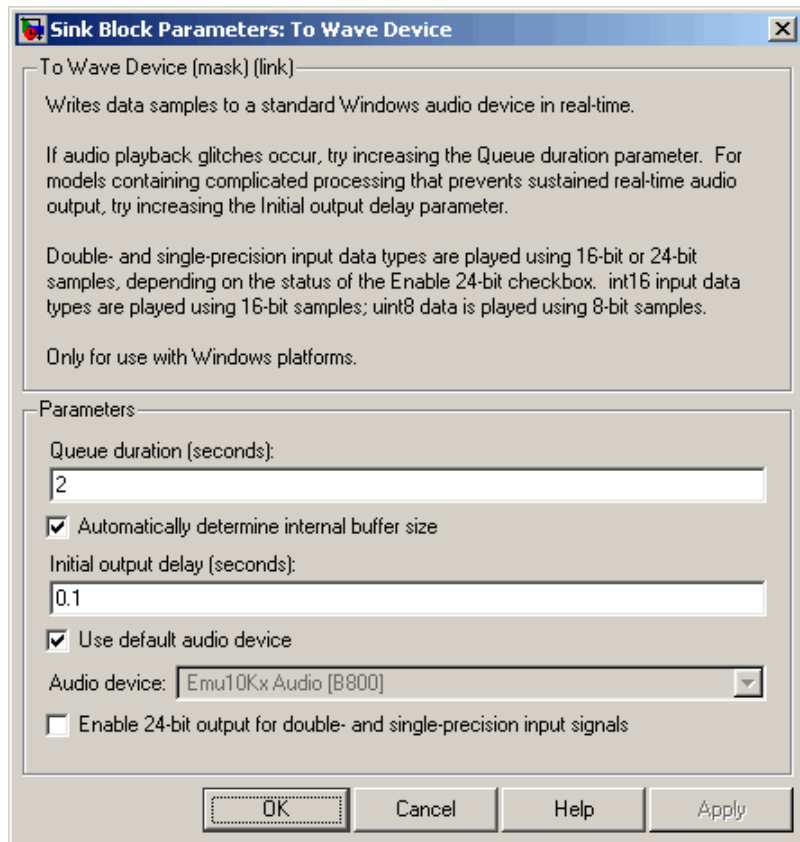
Queue duration (seconds)
> Specify the overall buffer size. To minimize the chance of dropouts, the block checks to make sure that the queue duration is as least as large as twice the internal buffer size. If it is not, the queue duration is automatically set to twice the internal buffer size.

Automatically determine internal buffer size
> Select to have the block automatically select the internal buffer size for you. For details, see "Buffering" on page 2-1548.

To Wave Device (Obsolete)

User-defined internal buffer size (samples)
Define the internal buffer size, or the size of the chunks of data sent by the block to the audio hardware device.

This parameter is only visible when **Automatically determine internal buffer size** is not selected.

Initial output delay (seconds)
Specify the amount of time by which to delay the initial output to the audio device. During this time data accumulates in the block's buffer. Any value less than or equal to the queue duration specifies the smallest possible initial delay, which is a single frame.

Use default audio device
Select to direct audio output to the system's default audio device.

Audio device
This parameter lists the names of the installed audio devices. Specify the name of the audio device to receive the audio output. Select **Use default audio device** when the system has only a single audio card installed.

This parameter is only enabled when the **Use default audio device** check box is not selected.

Enable 24-bit output for double and single precision input signals
Select to output 24-bit data when inputs are double- or single-precision. Otherwise, the block outputs 16-bit data for double- and single-precision inputs.

Port	Supported Data Types
Supported Data Types	
Input	• Double-precision floating point
	• Single-precision floating point
	• Signed fixed point with a word length of 16 and a fraction length of 15
	• 16-bit signed integers
	• 8-bit unsigned integers

See Also

From Wave Device (Obsolete)	DSP System Toolbox
To Wave File (Obsolete)	DSP System Toolbox
audioplayer	MATLAB
sound	MATLAB

To Wave File (Obsolete)

Purpose   Write audio data to file in Microsoft Wave (`.wav`) format

Library   dspwin32

Description

> **Note** The To Wave File block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the To Multimedia File block.

The To Wave File block streams audio data to a Microsoft Wave (`.wav`) file in the uncompressed pulse code modulation (PCM) format. For compatibility reasons, the sample rate of the discrete-time input signal should typically be one of the standard Windows audio device rates (8000, 11025, 22050, or 44100 Hz), although the block supports arbitrary rates.

The input to the block, *u*, can contain audio data with one or more channels. A signal with *C* channels is represented as a sample-based length-*C* vector or a frame-based *M*-by-*C* 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
```

> **Note** AVI files are the only supported file type for non-Windows platforms.

**Dialog
Box**
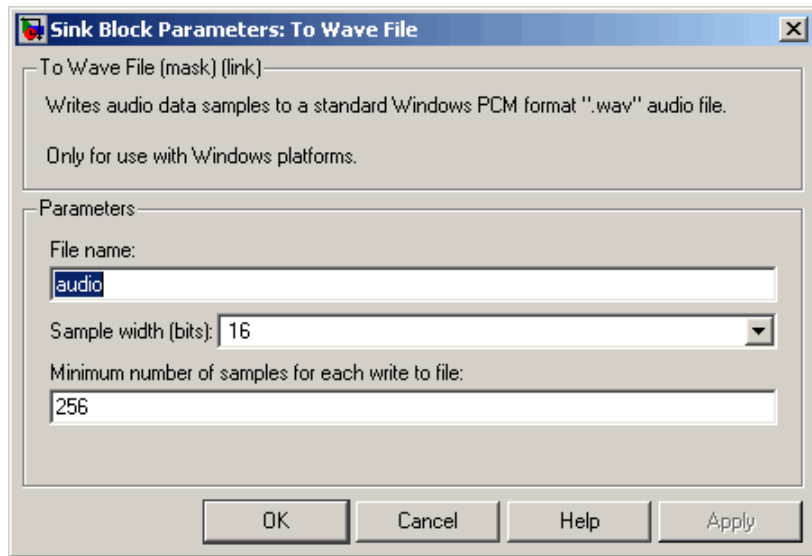
File name

Specify the path and name of the file to write. Paths can be relative or absolute. You do not need to specify the .wav extension.

Sample width (bits)

Specify the number of bits used to represent the signal samples in the file. The higher sample width settings require more memory but yield better fidelity for double- and single-precision inputs:

- 8 — Allocates 8 bits to each sample, allowing a resolution of 256 levels

- 16 — Allocates 16 bits to each sample, allowing a resolution of 65536 levels

- 24 — Allocates 24 bits to each sample, allowing a resolution of 16777216 levels

- 32 — Allocates 32 bits to each sample, allowing a resolution of 2^{32} levels ranging from -1 to 1

To Wave File (Obsolete)

The 8-, 16-, and 24-bit modes output integer data, while the 32-bit mode outputs single-precision floating-point data.

Minimum number of samples for each write to file

Specify the number of consecutive samples, L, to write with each file access. To reduce the required number of file accesses, the block writes L consecutive samples to the file during each access for $L \geq M$. 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.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Signed fixed point with a word length of 16 and a fraction length of 15
	• 16-bit signed integers
	• 8-bit unsigned integers

See Also

From Multimedia File	DSP System Toolbox
To Audio Device	DSP System Toolbox
Signal To Workspace	DSP System Toolbox
To Workspace	Simulink
wavwrite	MATLAB

Purpose   Transpose matrix

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

dspmtrx3
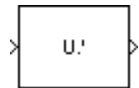
Description   The Transpose block transposes the M-by-N input matrix to size
N-by-M. When you select the **Hermitian** check box, the block performs
the Hermitian (complex conjugate) transpose.

```
y = u'     % Equivalent MATLAB code
```

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{bmatrix} \quad \xrightarrow{u'} \quad \begin{bmatrix} u_{11}^{*} & u_{21}^{*} \\ u_{12}^{*} & u_{22}^{*} \\ u_{13}^{*} & u_{23}^{*} \end{bmatrix}$$

When you do not select the **Hermitian** check box, the block performs
the nonconjugate transpose.

```
y = u.'     % Equivalent MATLAB code
```

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{bmatrix} \quad \xrightarrow{u.'} \quad \begin{bmatrix} u_{11} & u_{21} \\ u_{12} & u_{22} \\ u_{13} & u_{23} \end{bmatrix}$$

The block treats length-M vector input as an M-by-1 matrix.

The Transpose block supports real and complex floating-point and
fixed-point data types. When **Hermitian** is selected, the block input
must be a signed data type.

Transpose

**Dialog
Box**
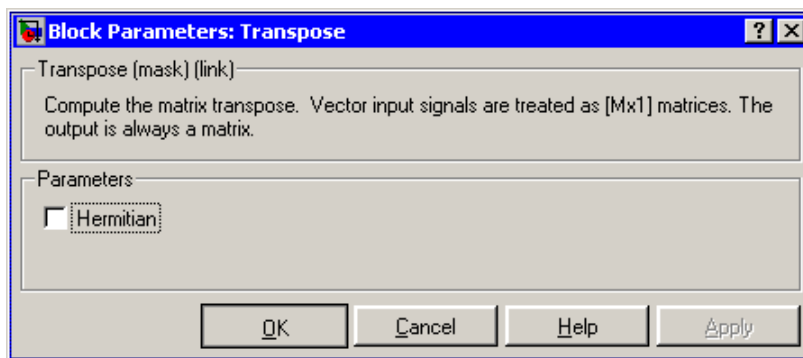
Hermitian
    When selected, specifies the complex conjugate transpose.

Saturate on integer overflow
    This parameter is only visible when the **Hermitian** parameter is
    selected because overflows can occur when computing the complex
    conjugate of complex fixed-point signals. When you select this
    parameter, such overflows saturate. This parameter is ignored for
    floating-point signals and for real-valued fixed-point signals.

**Supported
Data
Types**

When **Hermitian** is selected, the block input must be a signed data
type.

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Math Function	Simulink
Permute Matrix	DSP System Toolbox
Reshape	Simulink
Submatrix	DSP System Toolbox

Triggered Delay Line (Obsolete)

Purpose　　　　Buffer sequence of inputs into frame-based output

Library　　　　dspobslib

Description
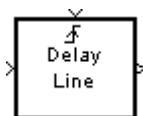
Note The Triggered Delay Line block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Delay Line block.

The Triggered Delay Line block acquires a collection of M_o input samples into a frame, where you specify M_o in the **Delay line size** parameter. The block buffers a single sample from input 1 whenever it is triggered by the control signal at input 2 (↯). When the next triggering event occurs, the newly acquired input sample is appended to the output frame so that the new output overlaps the previous output by M_o-1 samples. Between triggering events the block ignores input 1 and holds the output at its last value.

You specify the triggering event at input 2 in the **Trigger type** pop-up menu:

- Rising edge triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.

- Falling edge triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.

- Either edge triggers execution of the block when either a rising or falling edge (as described above) occurs.

The Triggered Delay Line block has zero latency, so the new input appears at the output in the same simulation time step. The output frame period is the same as the input sample period, $T_{fo}=T_{si}$.

Sample-Based Operation

In sample-based operation, the Triggered Delay Line block buffers a sequence of sample-based length-N vector inputs (1-D, row, or column) into a sequence of overlapping sample-based M_o-by-N matrix outputs, where you specify M_o in the **Delay line size** parameter ($M_o>1$). That is, each input vector becomes a *row* in the sample-based output matrix. When $M_o=1$, the input is simply passed through to the output, and retains the same dimension. Sample-based full-dimension matrix inputs are not accepted.

Frame-Based Operation

In frame-based operation, the Triggered Delay Line block rebuffers a sequence of frame-based M_i-by-N matrix inputs into an sequence of overlapping frame-based M_o-by-N matrix outputs, where M_o is the output frame size specified by the **Delay line size** parameter (that is, the number of consecutive samples from the input frame to rebuffer into the output frame). M_o can be greater or less than the input frame size, M_i. Each of the N input channels is rebuffered independently.

Initial Conditions

The Triggered Delay Line block's buffer is initialized to the value specified by the **Initial condition** parameter. The block always outputs this buffer at the first simulation step (*t*=0). When the block's output is a vector, the **Initial condition** can be a vector of the same size or a scalar value to be repeated across all elements of the initial output. When the block's output is a matrix, the **Initial condition** can be a matrix of the same size or a scalar to be repeated across all elements of the initial output.

Triggered Delay Line (Obsolete)

Dialog Box
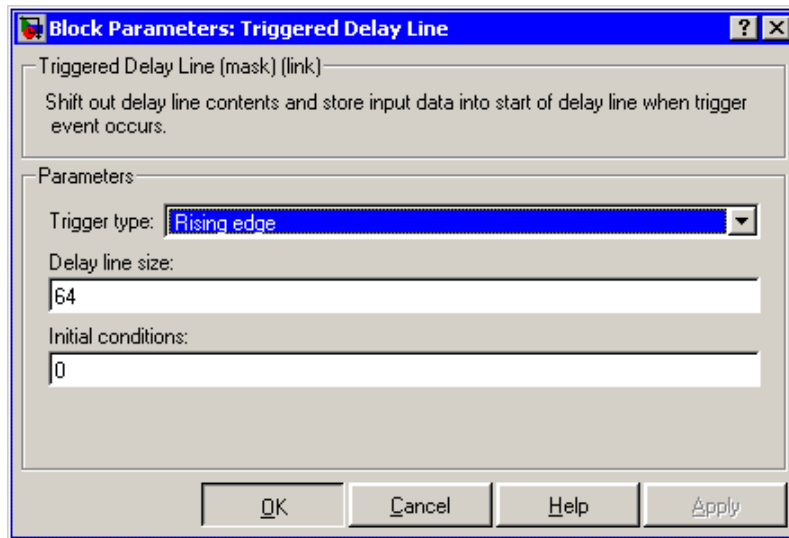
Trigger type
>   The type of event that triggers the block's execution.

Delay line size
>   The length of the output frame (number of rows in output matrix), M_o.

Initial condition
>   The value of the block's initial output, a scalar, vector, or matrix.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Trigger	• Any data type supported by the Trigger block
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also

Buffer	DSP System Toolbox
Delay Line	DSP System Toolbox
Unbuffer	DSP System Toolbox

Triggered Signal From Workspace

Purpose   Import signal samples from MATLAB workspace when triggered

Library   Signal Operations

dspsigops

Description

>$\digamma$ 1:10 ▷
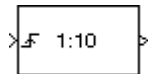
The Triggered Signal From Workspace block imports signal samples
from the MATLAB workspace into the Simulink model when triggered
by the control signal at the input port ($\digamma$). The **Signal** parameter
specifies the name of a MATLAB workspace variable containing the
signal to import, or any valid MATLAB expression defining a matrix or
3-D array.

When the **Signal** parameter specifies an M-by-N matrix (M≠1), each of
the N columns is treated as a distinct channel. You specify the frame
size in the **Samples per frame** parameter, M_o, and the output when
triggered is an M_o-by-N matrix containing M_o consecutive samples from
each signal channel. For M_o=1, the output is sample based; otherwise
the output is frame based. For convenience, an imported row vector
(M=1) is treated as a single channel, so the output dimension is M_o-by-1.

When the **Signal** parameter specifies an M-by-N-by-P array, the block
generates a single page of the array (an M-by-N matrix) at each trigger
time. The **Samples per frame** parameter must be set to 1, and the
output is always sample based.

Trigger Event

You specify the triggering event at the input port in the **Trigger type**
pop-up menu:

- Rising edge triggers execution of the block when the trigger input
  rises from a negative value to zero or a positive value, or from zero
  to a positive value.

- Falling edge triggers execution of the block when the trigger input
  falls from a positive value to zero or a negative value, or from zero
  to a negative value.

- `Either edge` triggers execution of the block when either a rising or falling edge (as described above) occurs.

Initial and Final Conditions

The **Initial output** parameter specifies the output of the block from the start of the simulation until the first trigger event arrives. Between trigger events, the block holds the output value constant at its most recent value (that is, no linear interpolation takes place). For single-channel signals, the **Initial output** parameter value can be a vector of length M_o or a scalar to repeat across the M_o elements of the initial output frames. For matrix outputs (M_o-by-N or M-by-N), the **Initial output** parameter value can be a matrix of the same size or a scalar to be repeated across all elements of the initial output.

When the block has output all of the available signal samples, it can start again at the beginning of the signal, or simply repeat the final value or generate zeros until the end of the simulation. (The block does not extrapolate the imported signal beyond the last sample.) The **Form output after final data value by** parameter controls this behavior:

- When you specify `Setting To Zero`, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.

- When you specify `Holding Final Value`, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.

- When you specify `Cyclic Repetition`, the block repeats the signal from the beginning after generating the last frame. When there are not enough samples at the end of the signal to fill the final frame, the block zero-pads the final frame as necessary to ensure that the output for each cycle is identical (for example, the *i*th frame of one cycle contains the same samples as the *i*th frame of any other cycle).

Triggered Signal From Workspace
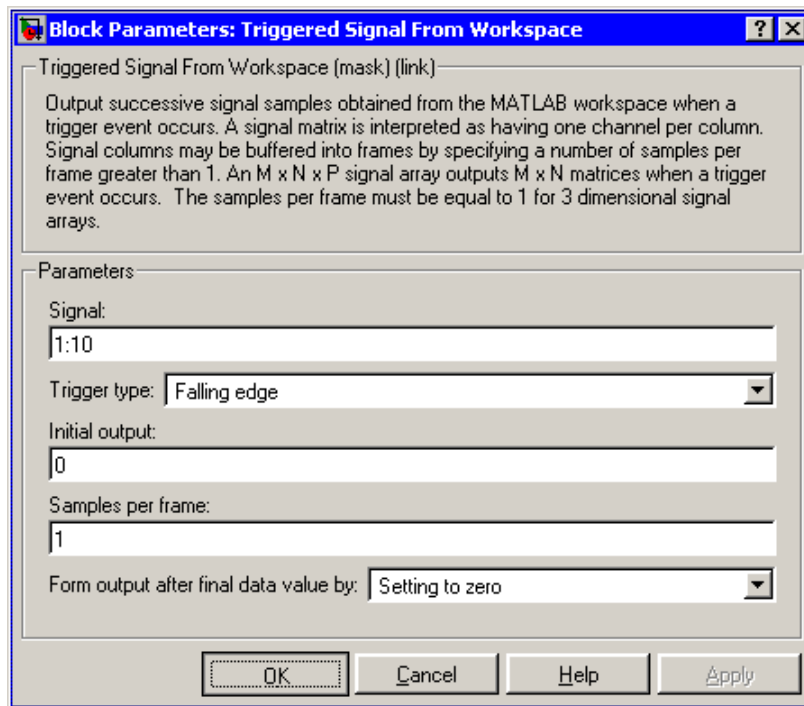
Dialog Box

Signal
The name of the MATLAB workspace variable from which to import the signal, or a valid MATLAB expression specifying the signal.

Trigger type
The type of event that triggers the block's execution.

Initial output
The value to output until the first trigger event is received.

Samples per frame
The number of samples, M_o, to buffer into each output frame. This value must be 1 when you specify a 3-D array in the **Signal** parameter.

Form output after final data value by
Specifies the output after all of the specified signal samples have been generated. The block can output zeros for the duration of the simulation (Setting to zero), repeat the final data sample (Holding Final Value) or repeat the entire signal from the beginning (Cyclic Repetition).

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

See Also

Signal From Workspace	DSP System Toolbox
Signal To Workspace	DSP System Toolbox
Triggered To Workspace	DSP System Toolbox

Triggered To Workspace

Purpose   Write input sample to MATLAB workspace when triggered

Library   Sinks

dspsnks4

Description
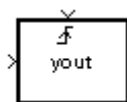
The Triggered To Workspace block creates a matrix or array variable in the MATLAB workspace, where it stores the acquired inputs at the end of a simulation. The block overwrites an existing variable with the same name.

When you set the **Save 2-D signals as** parameter to `2-D array` (`concatenate along first dimension`, the block saves an M-by-N input as a P-by-N matrix, where P is the **Maximum number of rows** parameter. When the simulation progresses long enough for the block to acquire more than P samples, the block stores only the most recent P samples. The **Decimation factor**, D, allows you to store only every Dth input matrix.

When you set the **Save 2-D signals as** parameter to `3-D array` (`concatenate along third dimension`, the block saves an M-by-N input as a three-dimensional array in which each M-by-N page represents a single sample from each of the $M*N$ channels (the most recent input matrix occupies the last page). The maximum size of this variable is limited to M-by-N-by-P, where P is the **Maximum number of rows** parameter. When the simulation progresses long enough for the block to acquire more than P inputs, it stores only the last P inputs. The **Decimation factor**, D, allows you to store only every Dth input matrix.

The block acquires and buffers a single frame from input 1 whenever it is triggered by the control signal at input 2 (�). At all other times, the block ignores input 1. You specify the triggering event at input 2 in the **Trigger type** pop-up menu:

- `Rising edge` triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.

- `Falling edge` triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.

- `Either edge` triggers execution of the block when either a rising or falling edge (as described above) occurs.

To save a record of the sample time corresponding to each sample value, open the Configuration Parameters dialog box. In the **Select** pane, click **Data Import/Export**. In the **Save to workspace** section, select the **Time** check box.

The nontriggered version of this block is the Signal To Workspace block.

Triggered To Workspace
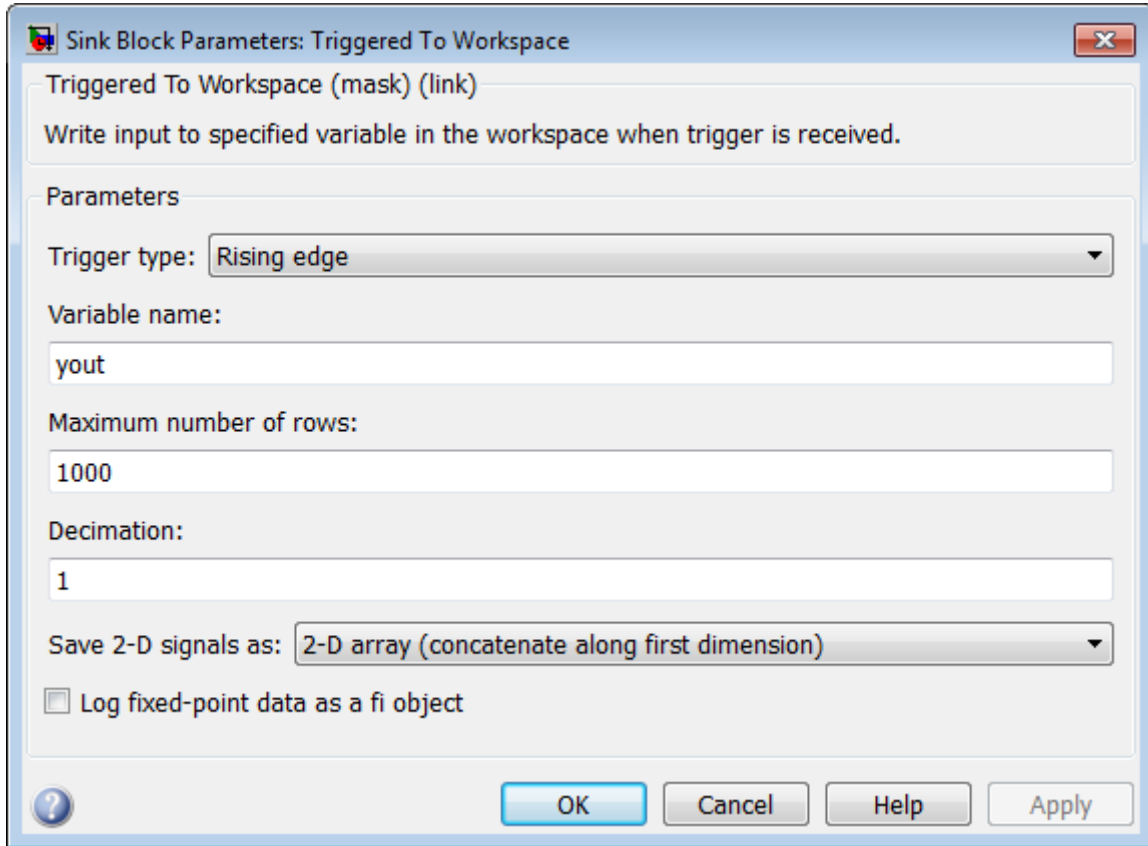
Trigger type
>   The type of event that triggers the block's execution.

Variable name
>   The name of the workspace variable in which to store the data.

Maximum number of rows

The maximum number of rows (one row per time step) to be saved, *P*.

Decimation

The decimation factor, *D*.

Save 2-D signals as

Specify whether the block saves 2-D signals as a 2-D or 3-D array in the MATLAB workspace:

- `2-D array (concatenate along first dimension)` — When you select this option, the block vertically concatenates each *M*-by-*N* matrix input with the previous input to produce a 2-D output array.

- `3-D array (concatenate along third dimension)` — When you select this option, the block saves an *M*-by-*N* input signal as a 3-D array. The maximum size of this 3-D array is limited to *M*-by-*N*-by-*P*, where *P* is the **Maximum number of rows** parameter. When the simulation progresses long enough for the block to acquire more than *P* inputs, the block stores only the last *P* inputs. The **Decimation factor**, *D*, allows you to store only every *D*th input matrix.

Note The `Inherit from input (this choice will be removed - see release notes)` option will be removed in a future release. See the *DSP System Toolbox Release Notes* for more information.

Log fixed-point data as a fi object

Select to log fixed-point data to the MATLAB workspace as a Fixed-Point Toolbox `fi` object. Otherwise, fixed-point data is logged to the workspace as `double`.

Triggered To Workspace

Supported Data Types

Port	Supported Data Types
Input	• Any data type supported by the To Workspace block
Trigger	• Any data type supported by the Trigger block

See Also

Signal From Workspace	DSP System Toolbox
Signal To Workspace	DSP System Toolbox
To Workspace	Simulink
Triggered Signal From Workspace	DSP System Toolbox

Purpose   Decompose signal into high-frequency and low-frequency subbands

Library   Filtering / Multirate Filters

dspmlti4
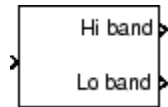
Description   The Two-Channel Analysis Subband Filter block decomposes the input into high-frequency and low-frequency subbands, each with half the bandwidth and half the sample rate of the input.

The block filters the input with a pair of highpass and lowpass FIR filters, and then downsamples the results by 2, as illustrated in the following figure.
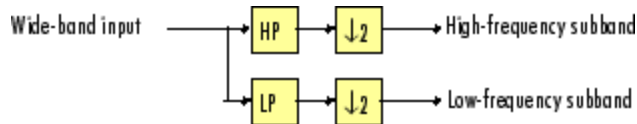
The block implements the FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than the straightforward filter-then-decimate algorithm shown in the preceding figure. Each subband is the first phase of the respective polyphase filter. You can implement a multilevel dyadic analysis filter bank by connecting multiple copies of this block or by using the Dyadic Analysis Filter Bank block. See "Creating Multilevel Dyadic Analysis Filter Banks" on page 2-1578 for more information.

You must provide a vector of filter coefficients for the lowpass and highpass FIR filters. Each filter should be a half-band filter that passes the frequency band that the other filter stops.

See the following topics for more information about this block:

- "Specifying the FIR Filters" on page 2-1576
- "Frame-Based Processing" on page 2-1576
- "Sample-Based Processing" on page 2-1577
- "Latency" on page 2-1577

• "Creating Multilevel Dyadic Analysis Filter Banks" on page 2-1578

Specifying the FIR Filters

You must provide the vector of numerator coefficients for the lowpass and highpass filters in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

For example, to specify a filter with the following transfer function, enter the vector `[b(1) b(2) ...  b(m)]`.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

Each filter should be a half-band filter that passes the frequency band that the other filter stops. You can use the Two-Channel Synthesis Subband Filter block to reconstruct the input to this block. To do so, you must design perfect reconstruction filters to use in the synthesis subband filter.

The best way to design perfect reconstruction filters is to use the Wavelet Toolbox `wfilters` function in to design both the filters both in this block and in the Two-Channel Synthesis Subband Filter block. You can also use other DSP System Toolbox and Signal Processing Toolbox functions. To learn how to design your own perfect reconstruction filters, see "References" on page 2-1589.

The Two-Channel Analysis Subband Filter block initializes all filter states to zero.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block accepts any two M-by-N matrices with the same frame rates. The block treats each column of the input as the high- or low-frequency subbands of the corresponding output channel. You can use the **Rate options** parameter to specify how the block resamples the input:

• When you set the **Rate options** parameter to `Enforce single-rate processing`, the input to the block can be an M-by-N matrix, where

M is a multiple of two. The block treats each column of the input as an independent channel and decomposes each channel over time. The block outputs two matrices, where each column of the output is the high- or low-frequency subband of the corresponding input column. To maintain the input sample rate, the block decreases the output frame size by a factor of two.

- When you set the **Rate options** parameter to `Allow multirate processing`, the block treats an M_i-by-N matrix input as N independent channels and decomposes each channel over time. The block outputs two M-by-N matrices, where each column of the output is the high- or low-frequency subband of the corresponding input column. The input and output frame *sizes* are the same, but the frame *rate* of the output is half that of the input. Thus, the overall sample rate of the output is half that of the input.

  In this mode, the block has one frame of latency, as described in the "Latency" on page 2-1577 section.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-N matrix input as $M \cdot N$ independent channels. The block decomposes each channel over time and outputs two M-by-N matrices whose sample rates are half the input sample rate. Each element in the output matrix is the high- or low-frequency subband output of the corresponding element of the input matrix.

Depending on the setting of your Simulink configuration parameters, the output may have one sample of latency, as described in the "Latency" on page 2-1577 section.

Latency

When you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`, the Two-Channel Analysis Subband Filter block always has zero-tasking latency. *Zero-tasking latency* means that

the block propagates the first input sample (received at time t=0) as the first output sample.

When you set the **Rate options** parameter to `Allow multirate processing`, the Two-Channel Analysis Subband Filter block may exhibit latency. The amount of latency depends on the setting of the **Input processing** parameter of this block, and the setting of the Simulink **Tasking mode for periodic sample times** configuration parameter. The following table summarizes the conditions that produce latency when the block is performing multirate processing.

Input processing	Tasking mode for periodic sample times	Latency
Elements as channels (sample based)	SingleTasking	None.
	MultiTasking or Auto	One sample. The first output sample in each channel always has a value of 0.
Columns as channels (frame based)	SingleTasking, MultiTasking or Auto	One frame. All samples in the first output frame have a value of 0.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" in the *DSP System Toolbox User's Guide* and "Scheduling" in the *Simulink Coder User's Guide*.

Creating Multilevel Dyadic Analysis Filter Banks

The Two-Channel Analysis Subband Filter block is the basic unit of a dyadic analysis filter bank. You can connect several of these blocks to implement an *n*-level filter bank, as illustrated in the following figure. For a review of dyadic analysis filter banks, see the Dyadic Analysis Filter Bank block reference page.

When you create a filter bank by connecting multiple copies of this block, the output values of the filter bank differ depending on whether there is latency. Though the output values differ, both sets of values are valid; the difference arises from changes in latency. See the "Latency" on page 2-1577 section for more information about when latency can occur in the Two-Channel Analysis Subband Filter block.

In some cases, rather than connecting several Two-Channel Analysis Subband Filter blocks, you can use the Dyadic Analysis Filter Bank block, which is faster and requires less memory. In particular, the Dyadic Analysis Filter Bank block is more efficient under the following conditions:

- The frame size of the signal you are decomposing is a multiple of 2^n.

- You are decomposing the signal into $n+1$ or 2^n subbands.

In all other cases, use Two-Channel Analysis Subband Filter blocks to implement your filter banks.

Two-Channel Analysis Subband Filter

3-Level Dyadic Analysis Filter Banks

Conceptual illustration

Two-Channel Analysis Subband Filter block implementation
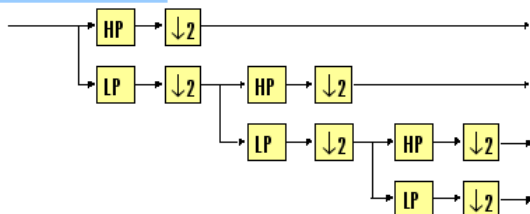
Both implementations of the dyadic analysis filter bank decompose a frame-based signal with frame size a multiple of 2^n into $n+1$ subbands, where $n = 3$.

In this case, the Dyadic Analysis Filter Bank block's implementation is more efficient.

Use the Two-Channel Analysis Subband Filter block implementation for other cases, such as to handle sample-based inputs, or to handle frame-based inputs whose frame size is not a multiple of 2^n.
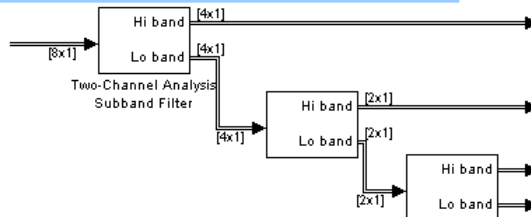
Dyadic Analysis Filter Bank block implementation
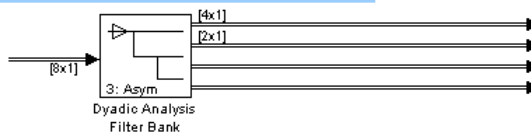
The Dyadic Analysis Filter Bank block allows you to specify the filter bank filters by providing vectors of filter coefficients, just as this block does. The Dyadic Analysis Filter Bank block provides an additional option of using wavelet-based filters that the block designs by using a wavelet you specify.

Fixed-Point Data Types

The Two-Channel Analysis Subband Filter Bank block is composed of two FIR Decimation blocks as shown in the following diagram.
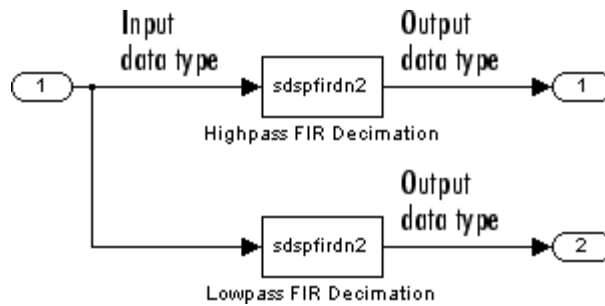
Input data type

Output data type

sdspfirdn2

Highpass FIR Decimation

Output data type

sdspfirdn2

Lowpass FIR Decimation

For fixed-point signals, you can set the coefficient, product output, accumulator, and output data types of the FIR Decimation blocks as discussed in "Dialog Box" on page 2-1581. For a diagram showing the usage of these data types, see the FIR Decimation block reference page.

Dialog Box

The **Main** pane of the Two-Channel Analysis Subband Filter block dialog appears as follows.

Two-Channel Analysis Subband Filter

Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in descending powers of z. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a third-order

Daubechies wavelet. When you use the Two-Channel Synthesis Subband Filter block to reconstruct the input to this block, you need to design perfect reconstruction filters to use in the synthesis subband filter. For more information, see "Specifying the FIR Filters" on page 2-1576.

Highpass FIR filter coefficients

Specify a vector of highpass FIR filter coefficients, in descending powers of *z*. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a third-order Daubechies wavelet. When you use the Two-Channel Synthesis Subband Filter block to reconstruct the input to this block, you need to design perfect reconstruction filters to use in the synthesis subband filter. For more information, see "Specifying the FIR Filters" on page 2-1576.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Note The Inherited (this choice will be removed - see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Two-Channel Analysis Subband Filter

Rate options

Specify the rate processing rule for the block. You can set this parameter to one of the following options:

- `Enforce single-rate processing` — When you select this option, the block treats each column of the input as an independent channel and decomposes each channel over time. The output has the same sample rate as the input, but the output frame size is half that of the input frame size. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.

- `Allow multirate processing` — When you select this option, the input and output of the block are the same size, but the sample rate of the output is half that of the input.

Some settings of this parameter cause the block to have nonzero latency. See "Latency" on page 2-1577 for more information.

The **Data Types** pane of the Two-Channel Analysis Subband Filter block dialog appears as follows.

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always rounded to Nearest.

> **Note** The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when all the following conditions exist:
>
> - **Product output data type** is Inherit: Inherit via internal rule
>
> - **Accumulator data type** is Inherit: Inherit via internal rule
>
> - **Output data type** is Inherit: Same as accumulator
>
> With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See "Fixed-Point Data Types" on page 2-1580 and "Multiplication Data Types" for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same word length as input

- An expression that evaluates to a valid data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button ⟨⟨ to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-1580 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

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

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-1580 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-1580 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets.* West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks.* Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks.* Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

DWT	DSP System Toolbox
Dyadic Analysis Filter Bank	DSP System Toolbox
FIR Decimation	DSP System Toolbox
IDWT	DSP System Toolbox
Two-Channel Synthesis Subband Filter	DSP System Toolbox
fir1	Signal Processing Toolbox
fir2	Signal Processing Toolbox
firls	Signal Processing Toolbox
wfilters	Wavelet Toolbox

Two-Channel Analysis Subband Filter

For related information, see "Multirate and Multistage Filters".

Purpose   Reconstruct signal from high-frequency and low-frequency subbands

Library   Filtering / Multirate Filters

dspmlti4

Description The Two-Channel Synthesis Subband Filter block reconstructs a signal from its high-frequency and low-frequency subbands, each with half the bandwidth and half the sample rate of the original signal. Use this block to reconstruct signals decomposed by the Two-Channel Analysis Subband Filter block.

The block upsamples the high- and low-frequency subbands by 2, and then filters the results with a pair of highpass and lowpass FIR filters, as illustrated in the following figure.
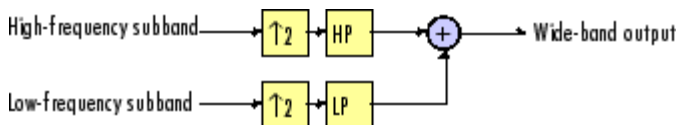
The block implements the FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than the straightforward interpolate-then-filter algorithm shown in the preceding figure. You can implement a multilevel dyadic synthesis filter bank by connecting multiple copies of this block or by using the Dyadic Synthesis Filter Bank block. For more information, see "Creating Multilevel Dyadic Synthesis Filter Banks" on page 2-1595.

You must provide a vector of filter coefficients for the lowpass and highpass FIR filters. Each filter should be a half-band filter that passes the frequency band that the other filter stops. You can use this block to reconstruct the output of a Two-Channel Analysis Subband Filter block. To do so, you must design the filters in this block such that they perfectly reconstruct the outputs of the analysis filters.

See the following topics for more information about this block:

• "Specifying the FIR Filters" on page 2-1592

Two-Channel Synthesis Subband Filter

- "Frame-Based Processing" on page 2-1592
- "Sample-Based Processing" on page 2-1593
- "Latency" on page 2-1594
- "Creating Multilevel Dyadic Synthesis Filter Banks" on page 2-1595

Specifying the FIR Filters

You must provide the vector of numerator coefficients for the lowpass and highpass filters in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

For example, to specify a filter with the following transfer function, enter the vector `[b(1) b(2) ...  b(m)]`.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

Each filter should be a half-band filter that passes the frequency band that the other filter stops. You can use this block to reconstruct the output of a Two-Channel Analysis Subband Filter block. To do so, you must design the filters in this block such that they perfectly reconstruct the outputs of the analysis filters.

The best way to design perfect reconstruction filters is to use the Wavelet Toolbox `wfilters` function for the filters in both this block *and* in the corresponding Two-Channel Analysis Subband Filter block. You can also use DSP System Toolbox and Signal Processing Toolbox functions. To learn how to design your own perfect reconstruction filters, see "References" on page 2-1605.

The Two-Channel Synthesis Subband Filter block initializes all filter states to zero.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block accepts any two *M*-by-*N* matrices with the same frame rates. The block treats each column of the input as the high- or low-frequency subbands of the corresponding output

channel. You can use the **Rate options** parameter to specify how the block resamples the input:

- When you set the **Rate options** parameter to `Enforce single-rate processing`, the input to the block can be any two *M*-by-*N* matrices with the same frame rate. The block treats each input column as the high- or low-frequency subbands of the corresponding output channel. The input to the topmost input port should contain the high-frequency subbands. The block outputs one matrix, where each column is reconstructed from the corresponding columns of each input matrix. The input and output frame *rates* are the same, but the frame *size* of the output is twice that of the input.

- When you set the **Rate options** parameter to `Allow multirate processing`, the block treats each column of the input as the high- or low-frequency subbands of the corresponding output channel. The input to the topmost input port should contain the high-frequency subbands. The block outputs one matrix, where each column is reconstructed from the corresponding columns of the input matrices. The input and output frame *sizes* are the same, but the frame *rate* of the output is twice that of the input. Thus, the overall sample rate of the output is twice that of the input sample rate.

  In this mode, the block has one frame of latency, as described in the "Latency" on page 2-1594 section.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block accepts any two *M*-by-*N* matrices with the same sample rates. The block treats each *M*-by-*N* matrix as *M* · *N* independent subbands. Each element of the input matrices is the high- or low-frequency subband of the corresponding channel in the output matrix. The input to the topmost input port should contain the high-frequency subbands. The block outputs one matrix with the same dimensions as the input matrices, but a sample rate that is twice that of the input. The block reconstructs each element of the output from the corresponding elements in the input matrices.

Depending on the setting of your Simulink configuration parameters, the output may have one sample of latency, as described in the "Latency" on page 2-1594 section.

Latency

When you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`, the Two-Channel Synthesis Subband Filter block always has zero-tasking latency. *Zero-tasking latency* means that the block propagates the first input sample (received at time $t = 0$) as the first output sample.

When you set the **Rate options** parameter to `Allow multirate processing`, the Two-Channel Synthesis Subband Filter block may exhibit latency. The amount of latency depends on the setting of the **Input processing** parameter of this block and the setting of the Simulink **Tasking mode for periodic sample times** configuration parameter. The following table summarizes the conditions that produce latency when the block is performing multirate processing.

Input processing	Tasking mode for periodic sample times	Latency
Elements as channels (sample based)	SingleTasking	None.
	MultiTasking or Auto	One sample. The first output sample in each channel always has a value of 0.
Columns as channels (frame based)	SingleTasking, MultiTasking or Auto	One frame. All samples in the first output frame have a value of 0.

> **Note** For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" in the *DSP System Toolbox User's Guide* and "Scheduling" in the *Simulink Coder User's Guide*.

Creating Multilevel Dyadic Synthesis Filter Banks

The Two-Channel Synthesis Subband Filter block is the basic unit of a dyadic synthesis filter bank. You can connect several of these blocks to implement an *n*-level filter bank, as illustrated in the following figure. For a review of dyadic synthesis filter banks, see the Dyadic Synthesis Filter Bank block reference page.

When you create a filter bank by connecting multiple copies of this block, the output values of the filter bank differ depending on whether there is latency. Though the output values differ, both sets of values are valid; the difference arises from changes in latency. See the "Latency" on page 2-1594 section for more information about when latency can occur in the Two-Channel Analysis Subband Filter block.

In some cases, rather than connecting several Two-Channel Analysis Subband Filter blocks, you can use the Dyadic Analysis Filter Bank block, which is faster and requires less memory. In particular, the Dyadic Analysis Filter Bank block is more efficient under the following conditions:

- You are reconstructing a signal from 2^n or $n+1$ subbands.

- The frame size of the signal you are reconstructing is a multiple of 2^n.

- The properties of the subbands you are working with match those of the outputs of the Dyadic Analysis Filter Bank block. These properties are described in the Dyadic Analysis Filter Bank reference page.

Two-Channel Synthesis Subband Filter

3-Level Dyadic Synthesis Filter Banks

Conceptual illustration

Two-Channel Synthesis Subband Filter block implementation
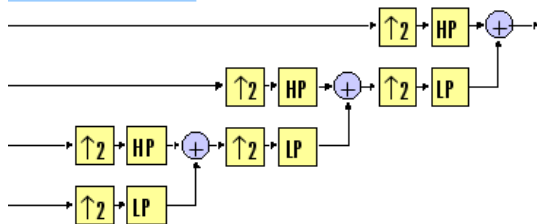
Both implementations of the dyadic analysis filter bank reconstruct a frame-based signal from n+1 subbands, where n = 3.

In this case, the Dyadic Synthesis Filter Bank block's implementation is more efficient, since the input subbands have the properties of the outputs of a Dyadic Analysis Filter Bank block.

Use the Two-Channel Synthesis Subband Filter block implementation for other cases, such as to handle separate sample-based vectors or matrices of subbands (rather than a single sample-based vector or matrix of concatenated subbands), or to output sample-based signals.
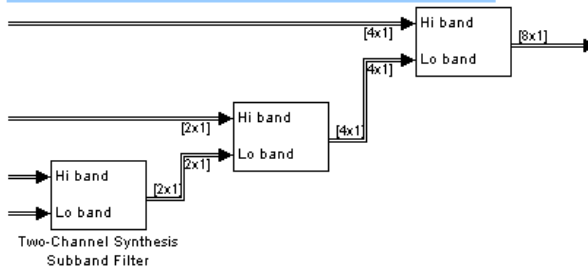
Dyadic Synthesis Filter Bank block implementation
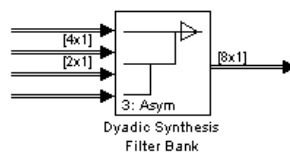
The Dyadic Synthesis Filter Bank block allows you to specify the filter bank filters by providing vectors of filter coefficients, just as this block does. The Dyadic Synthesis Filter Bank block provides an additional option of using wavelet-based filters that the block designs by using a wavelet you specify.

Fixed-Point Data Types

The Two-Channel Synthesis Subband Filter block is composed of two FIR Interpolation blocks as shown in the following diagram.
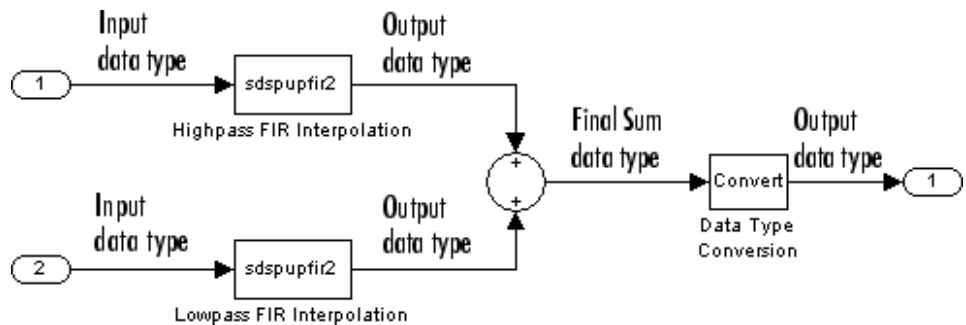
For fixed-point signals, you can set the coefficient, product output, accumulator, and output data types used in the FIR Interpolation blocks as discussed in "Dialog Box" on page 2-1597. For a diagram showing the usage of these data types within the FIR blocks, see the FIR Interpolation block reference page.

In addition, the inputs to the Sum block shown in the previous diagram are accumulated using the accumulator data type. The output of the Sum block is then cast from the accumulator data type to the output data type. Therefore the output of the Two-Channel Synthesis Subband Filter block is in the output data type. You also set these data types in the block dialog box as discussed in the "Dialog Box" on page 2-1597 section.

Dialog Box

The **Main** pane of the Two-Channel Synthesis Subband Filter block dialog appears as follows.

Two-Channel Synthesis Subband Filter
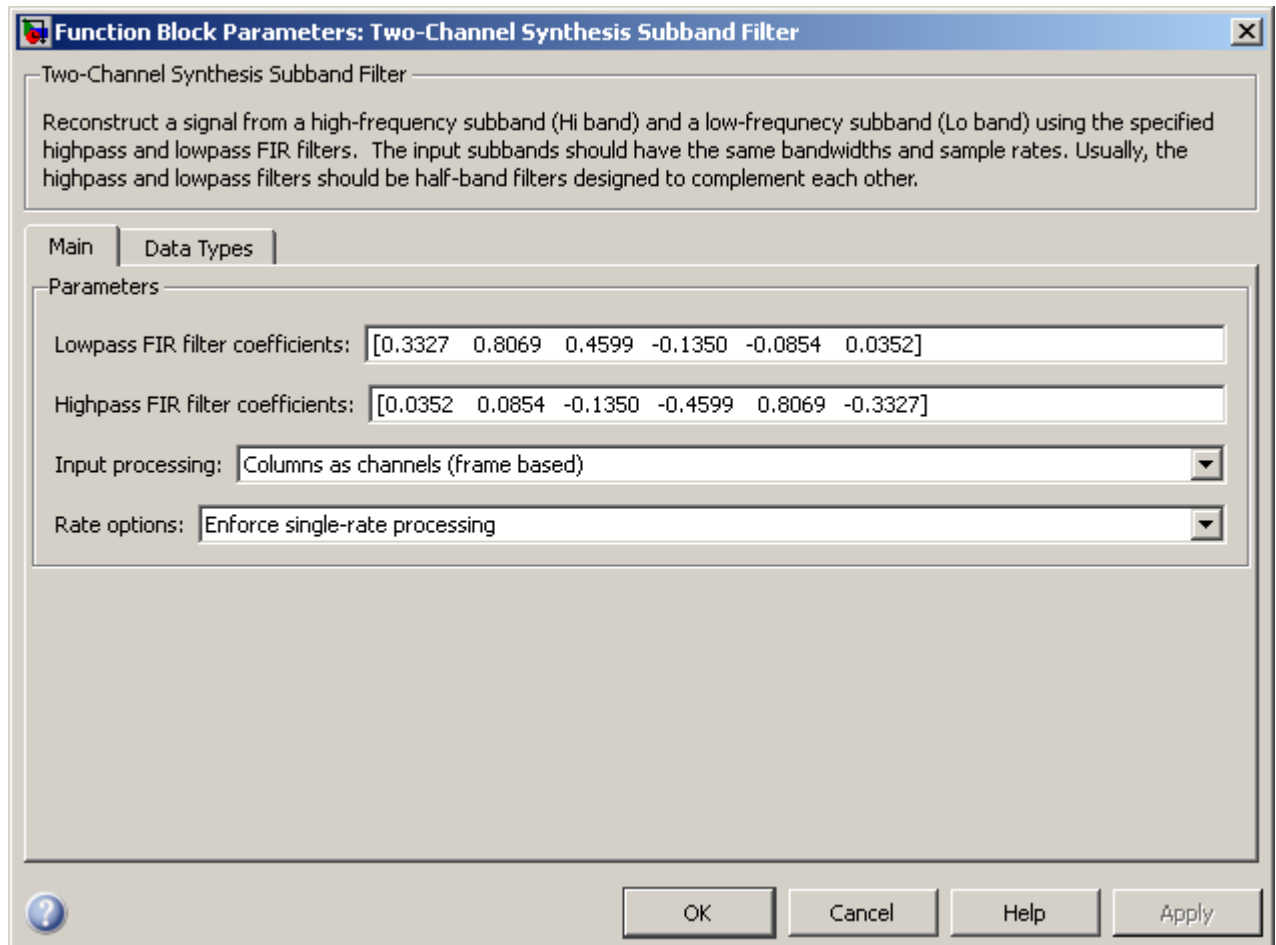
Lowpass FIR filter coefficients

A vector of lowpass FIR filter coefficients, in descending powers of *z*. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. To use this block to reconstruct the output of a Two-Channel Analysis

Subband Filter block, you must design the filters in this block to perfectly reconstruct the outputs of the analysis filters. For more information, see "Specifying the FIR Filters" on page 2-1592.

Highpass FIR filter coefficients

A vector of highpass FIR filter coefficients, in descending powers of *z*. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. To use this block to reconstruct the output of a Two-Channel Analysis Subband Filter block, you must design the filters in this block to perfectly reconstruct the outputs of the analysis filters. For more information, see "Specifying the FIR Filters" on page 2-1592.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Note The Inherited (this choice will be removed - see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

Specify the rate processing rule for the block. You can set this parameter to one of the following options:

- Enforce single-rate processing — When you select this option, the block treats each column of the input as an

independent channel and reconstructs each channel over time. The output has the same sample rate as the input, but the output frame size is twice that of the input frame size. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.

- `Allow multirate processing` — When you select this option, the input and output of the block are the same size, but the sample rate of the output is twice that of the input.

Some settings of this parameter cause the block to have nonzero latency. See "Latency" on page 2-1594 for more information.

The **Data Types** pane of the Two-Channel Synthesis Subband Filter block dialog appears as follows.

Round mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to Nearest.

> **Note** The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when all the following conditions exist:
>
> - **Product output data type** is `Inherit:  Inherit via internal rule`
>
> - **Accumulator data type** is `Inherit:  Inherit via internal rule`
>
> - **Output data type** is `Inherit:  Same as accumulator`
>
> With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients data type

Specify the coefficients data type. See "Fixed-Point Data Types" on page 2-1596 and "Multiplication Data Types" for illustrations depicting the use of the coefficients data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit:  Same word length as input`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button ⟩⟩ to display the **Data Type Assistant**, which helps you set the **Coefficients data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Product output data type

Specify the product output data type. See "Fixed-Point Data Types" on page 2-1596 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

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

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button `>>` to display the **Data Type Assistant**, which helps you set the **Product output data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Accumulator data type

Specify the accumulator data type. See "Fixed-Point Data Types" on page 2-1596 for illustrations depicting the use of the accumulator data type in this block. You can set this parameter to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Two-Channel Synthesis Subband Filter

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Accumulator data type** parameter.

See "Specify Data Types Using Data Type Assistant" in *Simulink User's Guide* for more information.

Output data type

Specify the output data type. See "Fixed-Point Data Types" on page 2-1596 for illustrations depicting the use of the output data type in this block. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as accumulator`

- An expression that evaluates to a valid data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide* for more information.

Minimum

Specify the minimum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Maximum

Specify the maximum value that the block should output. The default value is `[]` (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see "Signal Ranges")

- Automatic scaling of fixed-point data types

Lock data type settings against changes by the fixed-point tools
  Select this parameter to prevent the fixed-point tools from
  overriding the data types you specify on the block mask.

References   Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets.* West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks.* Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks.* Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types
- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

See Also

DWT	DSP System Toolbox
Dyadic Synthesis Filter Bank	DSP System Toolbox
FIR Interpolation	DSP System Toolbox
IDWT	DSP System Toolbox
Two-Channel Analysis Subband Filter	DSP System Toolbox
fir1	Signal Processing Toolbox
fir2	Signal Processing Toolbox
firls	Signal Processing Toolbox
wfilters	Wavelet Toolbox

Two-Channel Synthesis Subband Filter

For related information, see "Multirate and Multistage Filters".

Purpose   Receive `uint8` vector as UDP message

Library   Sources

dspsrcs4
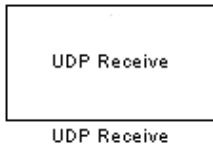
Description   The UDP Receive block receives UDP packets from an IP network port and saves them to its buffer. With each sample, the block output, emits the contents of a single UDP packet as a data vector.

```
┌─────────────────┐
│                 │
│  UDP Receive    │ ▷
│                 │
└─────────────────┘

    UDP Receive
```

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra .dll files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See `packNGo` for more information.

UDP Receive
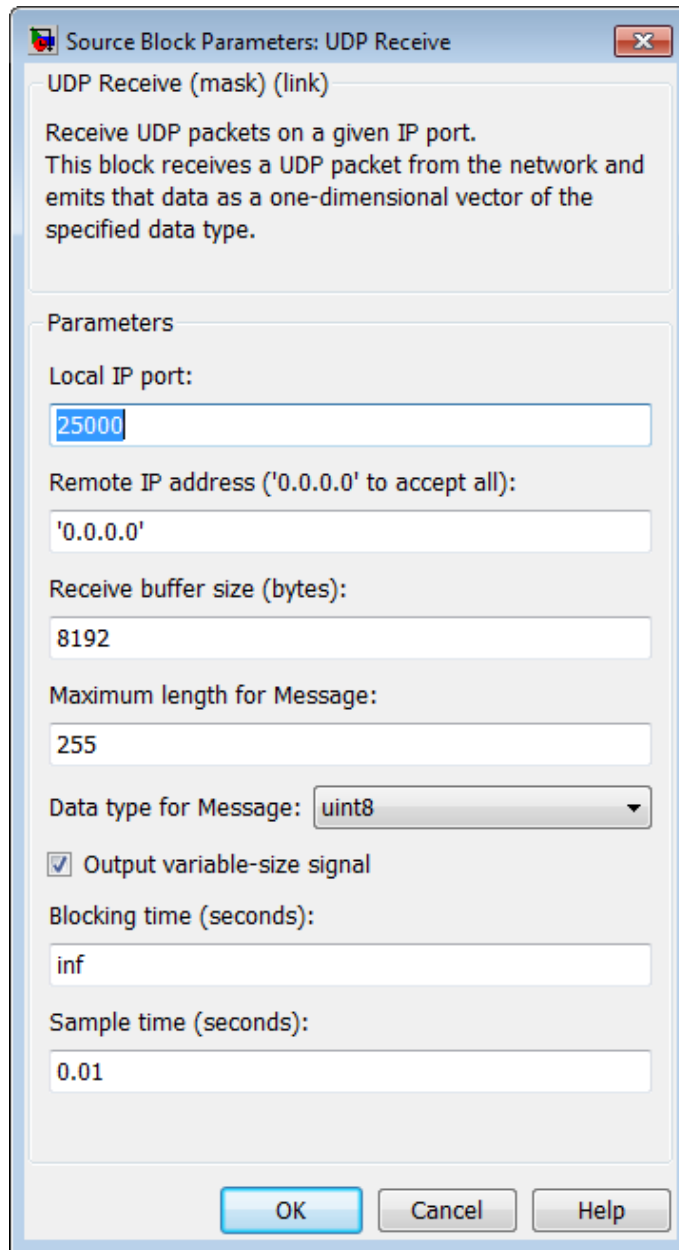
Dialog

Local IP port

> Specify the IP port number upon to receive UDP packets. This value defaults to 25000. The value can range 1–65535.

> **Note** On Linux, to set the IP port number below 1024, run
> MATLAB with root privileges. For example, at the Linux
> command line, enter:
>
> sudo matlab

Remote IP address ('0.0.0.0' to accept all)

Specify the IP address from which to accept packets. Entering a
specific IP address blocks UDP packets from other addresses. To
accept packets from any IP address, enter '0.0.0.0'. This value
defaults to '0.0.0.0'.

Receive buffer size (bytes)

Make the receive buffer large enough to avoid data loss caused by
buffer overflows. This value defaults to 8192.

Maximum length for Message

Specify the maximum length, in vector elements, of the data
output vector. Set this parameter to a value equal or greater than
the data size of a UDP packet. The system truncates data that
exceeds this length. This value defaults to 255.

If you disable **Output variable-size signal**, the block outputs
a fixed-length output the same length as the **Maximum length
for Message**.

Data type for Message

Set the data type of the vector elements in the Message output.
Match the data type with the data input used to create the UDP
packets. This option defaults to uint8.

Output variable-size signal

If your model supports signals of varying length, enable the
Output variable-size signal parameter. This checkbox defaults
to selected (enabled). In that case:

- The output vector varies in length, depending on the amount of data in the UDP packet.

- The block emits the data vector from a single unlabeled output.

If your model does not support signals of varying length, disable the **Output variable-size signal** parameter. In that case:

- The block emits a fixed-length output the same length as the **Maximum length for Message**.

- If the UDP packet contains less data than the fixed-length output, the difference contains invalid data.

- The block emits the data vector from the **Message** output.

- The block emits the length of the valid data from the **Length** output.

- The block dialog box displays the **Data type for Length** parameter.

In both cases, the block truncates data that exceeds the **Maximum length for Message**.

Data type for Length

Set the data type of the Length output. This option defaults to `double`.

Blocking time (seconds)

For each sample, wait this length of time for a UDP packet before returning control to the scheduler. This value defaults to `inf`, which indicates to wait indefinitely.

Note This parameter appears only in the Embedded Coder® UDP Receive block.

Sample time (seconds)

Specify how often the scheduler runs this block. Enter a value greater than zero. In real-time operation, setting this option to a

large value reduces the likelihood of dropped UDP messages. This value defaults to a sample time of 0.01 s.

See Also   Byte Pack, Byte Reversal, Byte Unpack, UDP Send

UDP Send

Purpose   Send UDP message

Library   Sinks

dspsnks4
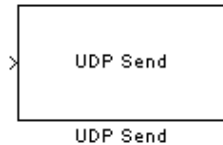
Description

UDP Send

The UDP Send block transmits an input vector as a UDP message over an IP network port.

The generated code for this block relies on prebuilt .dll files. You can run this code outside the MATLAB environment, or redeploy it, but be sure to account for these extra .dll files when doing so. The packNGo function creates a single zip file containing all of the pieces required to run or rebuild this code. See packNGo for more information.

Sink Block Parameters: UDP Send ✕

UDP Send (mask) (link)

Send a UDP packet to a network address identified by the remote IP address and remote IP port parameters.

Parameters

Remote IP address ('255.255.255.255' for broadcast):

| 127.0.0.1'

Remote IP port:

| 25000

Local IP port source: | Automatically determine ▾ |

OK   Cancel   Help   Apply

Dialog Box
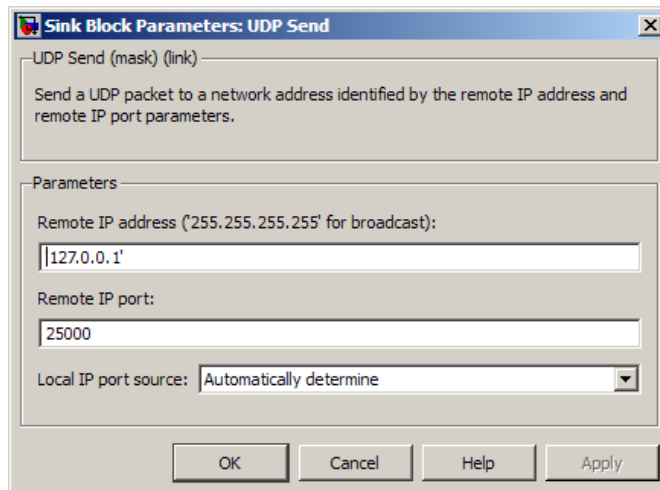
IP address ('255.255.255.255' for broadcast)
Specify the IP address or hostname to which the block sends the message. To broadcast the UDP message, retain the default value, '255.255.255.255'.

Remote IP port
Specify the port to which the block sends the message. The value defaults to 25000, but the values range from 1–65535.

Note On Linux, to set the IP port number below 1024, run MATLAB with root privileges. For example, at the Linux command line, enter:

```
sudo matlab
```

UDP Send

Local IP port source

To let the system automatically assign the port number, select `Assign automatically`. To specify the IP port number using the **Local IP port** parameter, select `Specify`.

Local IP port

Specify the IP port number from which the block sends the message.

If the receiving address expects messages from a particular port number, enter that number here.

See Also   Byte PackByte Reversal, Byte Unpack, UDP Receive

Purpose   Unbuffer input frame into sequence of scalar outputs

Library   Signal Management / Buffers

dspbuff3
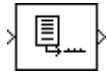
Description   The Unbuffer block unbuffers an M_i-by-N input into a 1-by-N output.
That is, inputs are unbuffered *row-wise* so that each matrix row
becomes an independent time-sample in the output. The rate at which
the block receives inputs is generally less than the rate at which the
block produces outputs.

"slow-time" input
(frame size = 3, frame period = 3*T_{si})

"fast-time" output
(frame size = 1, sample period = T_{si})
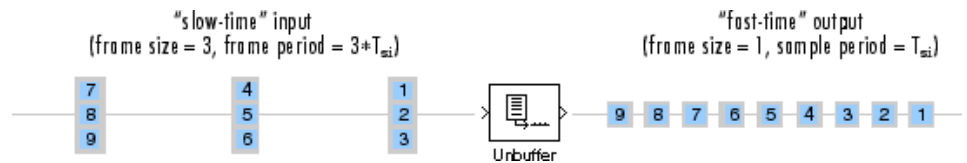
The block adjusts the output rate so that the *sample period* is the same
at both the input and output, $T_{so}=T_{si}$. Therefore, the output sample
period for an input of frame size M_i and frame period T_{fi} is T_{fi}/M_i, which
represents a *rate M_i* times higher than the input frame rate. In the
example above, the block receives inputs only once every three sample
periods, but produces an output once every sample period. To rebuffer
inputs to a larger or smaller frame size, use the Buffer block.

In the model below, the block unbuffers a four-channel input with a
frame size of three. The **Initial conditions** parameter is set to zero
and the tasking mode is set to multitasking, so the first three outputs
are zero vectors.

Unbuffer
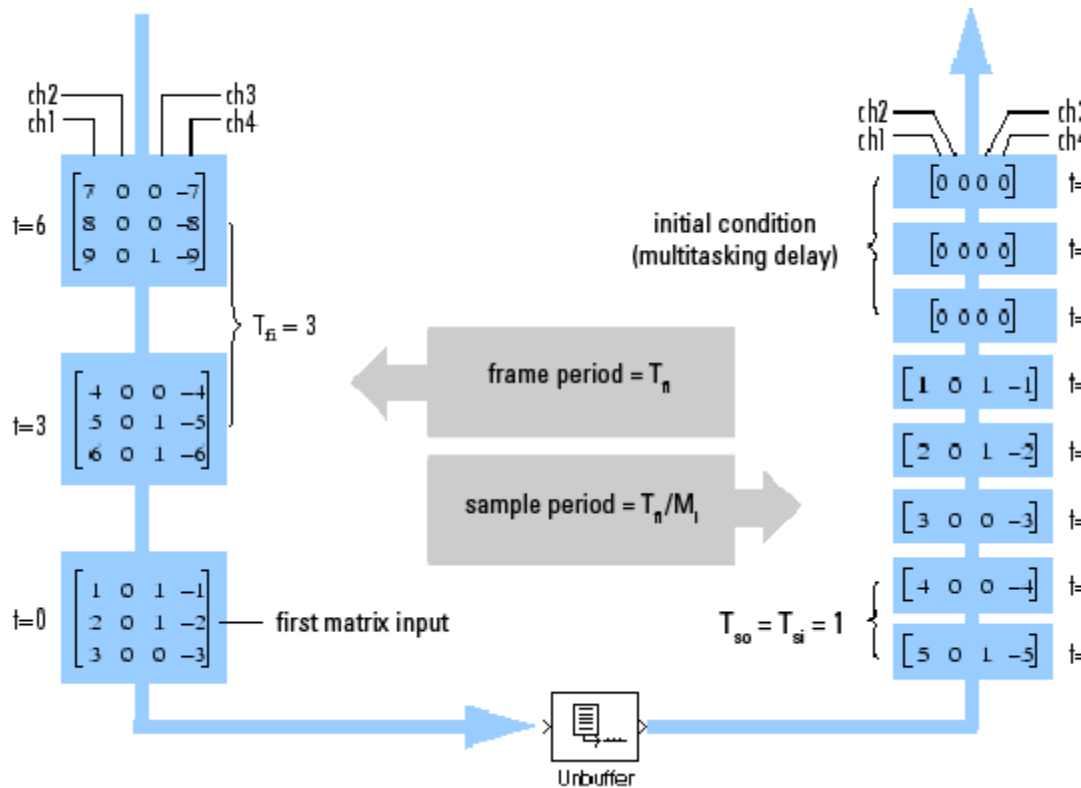
Zero Latency

The Unbuffer block has *zero-tasking latency* in Simulink single-tasking mode. Zero-tasking latency means that the first input sample (received at t=0) appears as the first output sample.

Nonzero Latency

For *multitasking* operation, the Unbuffer block's buffer is initialized with the value specified by the **Initial conditions** 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 conditions** parameter can be one of the following:

- A scalar to be repeated for the first M_i output samples of every channel

- A length-M_i vector containing the values of the first M_i output samples for every channel

- An M_i-by-N matrix containing the values of the first M_i output samples in each of N channels

---

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

Dialog Box

Function Block Parameters: Unbuffer	✕
Unbuffer	
Convert a frame to scalar samples output at a higher rate.	
Parameters	
Initial conditions:	
0	
Sample-based processing: Same as frame based ▾	
⊙	OK Cancel Help Apply
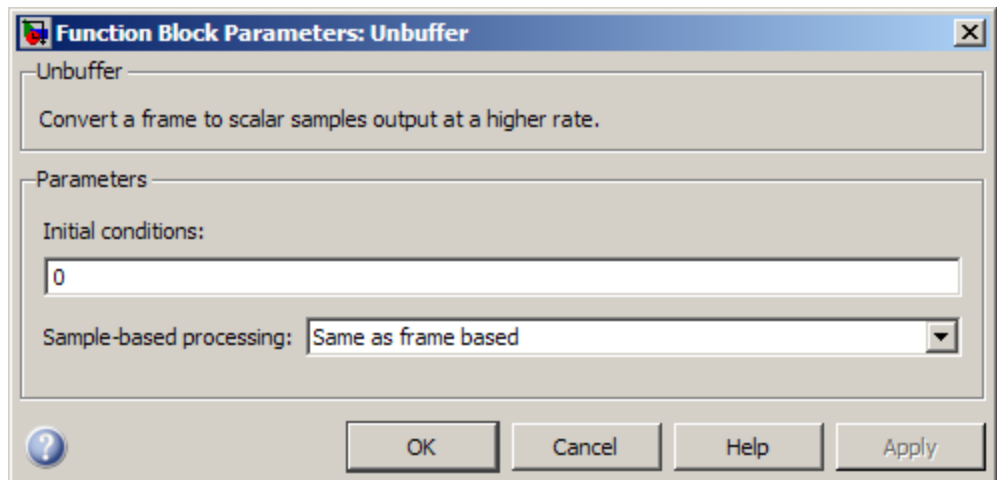
Initial conditions
The value of the block's initial output for cases of nonzero latency. You can specify a scalar, vector, or matrix.

Unbuffer

Sample-based processing

Specify how the block processes sample-based inputs. You can select one of the following options:

- Same as frame based — When you select this option, the block performs frame-based processing on sample-based inputs. The block unbuffers M_i-by-N matrix inputs into a 1-by-N output. Each row of the input matrix becomes an independent time-sample in the output.

- Pass through (this choice will be removed   see release notes) — When you select this option and the input to the block is sample based, the block does not unbuffer the input. The output of the block is the same as the input.

Note This parameter will be removed in a future release.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

See Also Buffer   DSP System Toolbox

See "Unbuffer Frame-Based Signals into Sample-Based Signals" for
related information.

Uniform Decoder

Purpose   Decode integer input into floating-point output

Library   Quantizers

dspquant2

Description
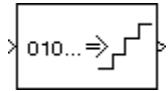
The Uniform Decoder block performs the inverse operation of the Uniform Encoder block, and reconstructs quantized floating-point values from encoded integer input. The block adheres to the definition for uniform decoding specified in ITU-T Recommendation G.701.

Inputs can be real or complex values of the following six integer data types: uint8, uint16, uint32, int8, int16, or int32.

The block first casts the integer input values to floating-point values, and then uniquely maps (decodes) them to one of 2^B uniformly spaced floating-point values in the range [-V, $(1-2^{1-B})$V], where you specify B in the **Bits** parameter (as an integer between 2 and 32) and V is a floating-point value specified by the **Peak** parameter. The smallest input value representable by B bits (0 for an unsigned input data type; -2^{B-1} for a signed input data type) is mapped to the value -V. The largest input value representable by B bits (2^B-1 for an unsigned input data type; $2^{B-1}-1$ for a signed input data type) is mapped to the value $(1-2^{1-B})$V. Intermediate input values are linearly mapped to the intermediate values in the range [-V, $(1-2^{1-B})$V].

To correctly decode values encoded by the Uniform Encoder block, the **Bits** and **Peak** parameters of the Uniform Decoder block should be set to the same values as the **Bits** and **Peak** parameters of the Uniform Encoder block. The **Overflow mode** parameter specifies the Uniform Decoder block's behavior when the integer input is outside the range representable by B bits. When you select **Saturate**, *unsigned* input values greater than 2^B-1 saturate at 2^B-1; *signed* input values greater than $2^{B-1}-1$ or less than -2^{B-1} saturate at those limits. The real and imaginary components of complex inputs saturate independently.

When you select **Wrap**, *unsigned* input values, u, greater than 2^B-1 are wrapped back into the range [0, 2^B-1] using mod-2^B arithmetic.

```
u = mod(u,2^B)
```

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))
```

The real and imaginary components of complex inputs wrap independently.

The **Output type** parameter specifies whether the decoded floating-point output is single or double precision. Either level of output precision can be used with any of the six integer input data types.

Examples See example model ex_uniform_decoder.

In this example, the input to the block is the uint8 output of a Uniform Encoder block. This block has comparable settings: **Peak** = 2, **Bits** = 3, and **Output type** = Unsigned. (Comparable settings ensure that inputs to the Uniform Decoder block do not saturate or wrap. See the example on the Uniform Encoder block reference page for more about these settings.)

The real and complex components of each input are independently mapped to one of 2^3 distinct levels in the range [-2.0,1.5].

```
0  is mapped to  -2.0
1  is mapped to  -1.5
2  is mapped to  -1.0
3  is mapped to  -0.5
4  is mapped to   0.0
5  is mapped to   0.5
6  is mapped to   1.0
7  is mapped to   1.5
```

Uniform Decoder

Dialog Box
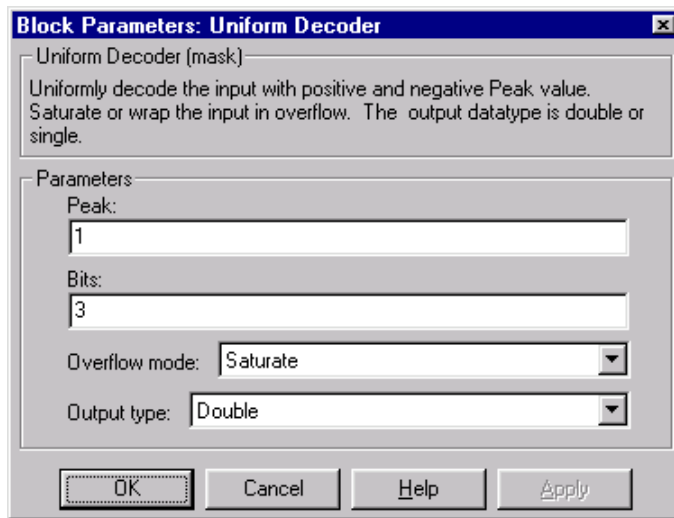
Peak

Specify 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

Specify 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

Specify 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

Specify the precision of the floating-point output, single or double.

References *General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms,* International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

Supported Data Types

Port	Supported Data Types
Input	• 8-, 16-, and 32-bit integers
Output	• Double-precision floating point
	• Single-precision floating point

See Also

Data Type Conversion	Simulink
Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
udecode	Signal Processing Toolbox
uencode	Signal Processing Toolbox

Uniform Encoder

Purpose   Quantize and encode floating-point input into integer output

Library   Quantizers

dspquant2
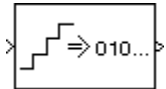
Description   The Uniform Encoder block performs the following two operations on
each floating-point sample in the input vector or matrix:

1 Quantizes the value using the same precision

2 Encodes the quantized floating-point value to an integer value

In the first step, the block quantizes an input value to one of 2^B
uniformly spaced levels in the range [-V, $(1-2^{1-B})V$], where you specify
B in the **Bits** parameter and you specify V in the **Peak** parameter.
The quantization process rounds both positive and negative inputs
downward to the nearest quantization level, with the exception of those
that fall exactly on a quantization boundary. The real and imaginary
components of complex inputs are quantized independently.

The number of bits, B, can be any integer value between 2 and 32,
inclusive. Inputs greater than $(1-2^{1-B})V$ or less than -V saturate at those
respective values. The real and imaginary components of complex
inputs saturate independently.

In the second step, the quantized floating-point value is uniquely
mapped (encoded) to one of 2^B integer values. When the **Output type** is
set to Unsigned integer, the smallest quantized floating-point value,
-V, is mapped to the integer 0, and the largest quantized floating-point
value, $(1-2^{1-B})V$, is mapped to the integer 2^B-1. Intermediate
quantized floating-point values are linearly (uniformly) mapped to the
intermediate integers in the range [0, 2^B-1]. For efficiency, the block
automatically selects an *unsigned* output data type (uint8, uint16, or
uint32) with the minimum number of bits equal to or greater than B.

When the **Output type** is set to Signed integer, the smallest
quantized floating-point value, -V, is mapped to the integer -2^{B-1}, and
the largest quantized floating-point value, $(1-2^{1-B})V$, is mapped to the

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 DSP System Toolbox blocks accept only double-precision inputs. Use the Simulink Data Type Conversion block to convert integer data types to double precision. See "Data Types" in the Simulink documentation for a complete discussion of data types, as well as a list of Simulink blocks capable of reduced-precision operations.

Bits	Unsigned Integer	Signed Integer
2 to 8	`uint8`	`int8`
9 to 16	`uint16`	`int16`
17 to 32	`uint32`	`int32`

The Uniform Encoder block operations adhere to the definition for uniform encoding specified in ITU-T Recommendation G.701.

Examples   See example model ex_uniform_encoder.

In this example, the following parameters are set:

- **Peak** = 2

- **Bits** = 3

- **Output type** = Unsigned

The following figure illustrates uniform encoding.

Uniform Encoder
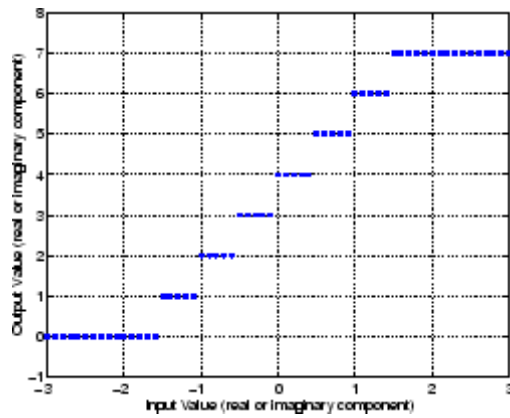
The real and complex components of each input (horizontal axis) are independently quantized to one of 2^3 distinct levels in the range [-2,1.5]. These components are then mapped to one of 2^3 integer values in the range [0,7].

```
-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
```

This table 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)

Input	Quantized Input	Output	Notes
0.4-1.2i	0.0-1.5i	4+i	
0.4-6.0i	0.0-2.0i	4	Saturation (imaginary)
-4.2+3.5i	-2.0+2.0i	7i	Saturation (real and imaginary)

The output data type is automatically set to uint8, the most efficient format for this input range.

Block Parameters: Uniform Encoder

Uniform Encoder (mask)

Uniformly quantize and encode the input into specified number of bits. The input is saturated at positive and negative Peak value. Output datatype is either 8, 16, or 32-bit signed or unsigned integer, based on the least number of bits needed.

Parameters

Peak:

1

Bits:

8

Output type: Unsigned integer

| OK | Cancel | Help | Apply |

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.

Uniform Encoder

Bits

Specify the number of bits, *B*, needed to represent the integer output. The number of levels at which the block quantizes the floating-point input is 2^B.

Output type

The data type of the block's output, `Unsigned integer` or `Signed integer`. Unsigned outputs are `uint8`, `uint16`, or `uint32`, while signed outputs are `int8`, `int16`, or `int32`.

References *General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms,* International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• 8-, 16-, and 32-bit integers

See Also

Data Type Conversion	Simulink
Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox
udecode	Signal Processing Toolbox
uencode	Signal Processing Toolbox

Purpose   Unwrap signal phase

Library   Signal Operations

                dspsigops

Description   The Unwrap block unwraps each channel of the input by adding or
                    subtracting appropriate multiples of 2π to each channel element. The
                    input can be a vector or matrix, and must have radian phase entries.
                    The block recognizes phase discontinuities larger than the **Tolerance**
                    parameter setting. For more information about phase unwrapping, see
                    the "Definition of Phase Unwrap" on page 2-1633.

                    The block preserves the input size and dimension, and the output port
                    rate equals the input port rate.

Unwrap Method

The Unwrap block unwraps each channel of its input matrix or input
vector by adding 2π*k* to each successive channel element, and updating
k at each *phase jump*. A phase jump occurs when the difference between
two adjacent phase value entries exceeds the value of the **Tolerance**
parameter.

The following code illustrates how the block unwraps the data in a
given input channel u.

```
k=0;    % initialize k to 0
i=1;    % initialize the counter to 1
alpha=pi;   % set alpha to the desired Tolerance. In this case, pi

for i = 1:(size(u)-1)
    yout(i,:)=u(i)+(2*pi*k); % add 2*pi*k to ui
    if((abs(u(i+1)-u(i)))>(abs(alpha)))  %if diff is greater than alph

        if u(i+1)<u(i)   % if the phase jump is negative, increment k
            k=k+1;
        else              % if the phase jump is positive, decrement k
            k=k-1;
```

```
            end
        end
end
yout((i+1),:)=u(i+1)+(2*pi*k); % add 2*pi*k to the last element of the in
```

Frame-Based Processing

When you configure the block to perform frame-based processing, the block supports two different unwrap modes. In both modes, the block adds $2\pi k$ to each input channel's elements, and updates k at each phase discontinuity. The difference between the two modes is how often the block resets the initial phase value (k) to zero. You can choose to unwrap data across frame boundaries (default), or to unwrap only within input frames, by resetting the initial phase value each time a new input frame is received.

Unwrapping Across Frame Boundaries

In the default mode, the block ignores boundaries between input frames, and continues to unwrap the data in each channel without resetting the initial phase value to zero. To specify this mode, clear the **Do not unwrap phase discontinuities between successive frames** check box. The following figure illustrates how the block unwraps data in this mode.

Default Frame-Based Unwrap Mode

☐ Do not unwrap phase discontinuities between successive frames
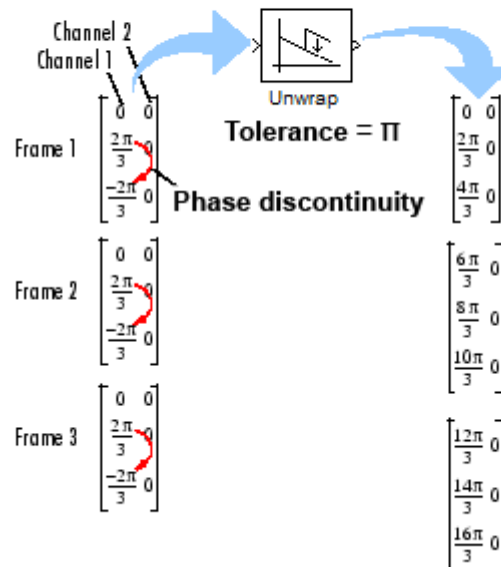
Unwrapping Within Frames

When you select the **Do not unwrap phase discontinuities between successive frames** check box, the block treats each frame of input data independently. In this mode, the block resets the initial phase value to zero each time a new input frame is received. The following figure illustrates how the block unwraps data in this mode.

Unwrap

Nondefault Frame-Based Unwrap Mode
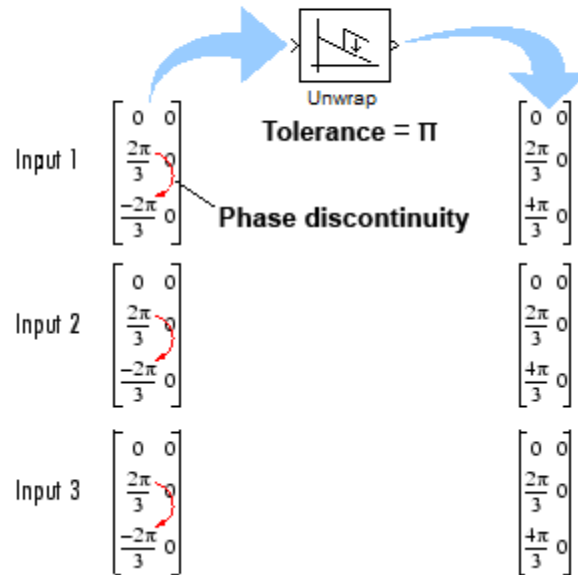
Sample-Based Processing

When you configure the block to perform sample-based processing, the block treats each element of the input as an individual channel. The block unwraps the data in each channel of the input, and does not reset the initial phase to zero each time a new input is received. The following figure illustrates how the block unwraps data when performing sample-based processing.
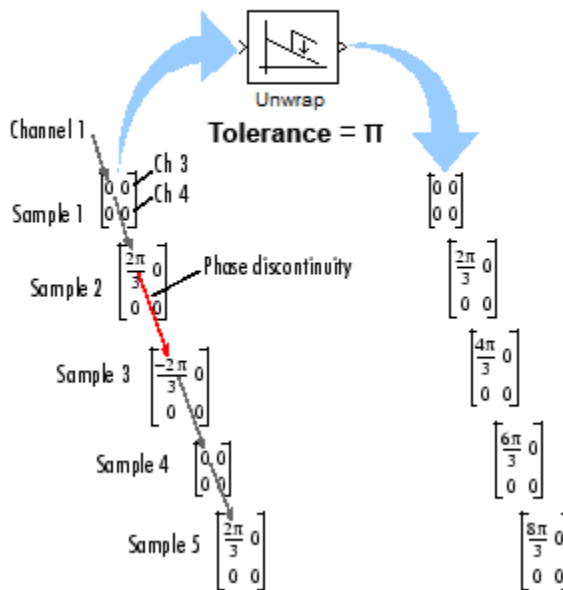
Definition of Phase Unwrap

Algorithms that compute the phase of a signal often only output phases between $-\pi$ and π. For instance, such algorithms compute the phase of $\sin(2\pi + 3)$ to be 3, since $\sin(3) = \sin(2\pi + 3)$, and since the actual phase, $2\pi + 3$, is not between $-\pi$ and π. Such algorithms compute the phases of $\sin(-4\pi + 3)$ and $\sin(16\pi + 3)$ to be 3 as well.

Phase unwrap or unwrap is a process often used to reconstruct a signal's original phase. Unwrap algorithms add appropriate multiples of 2π to each phase input to restore original phase values, as illustrated in the following diagram. See "Unwrap Method" on page 2-1629 for more information on the unwrap algorithm used by this block.

The following figure illustrates the concept of phase unwrapping.

Unwrap
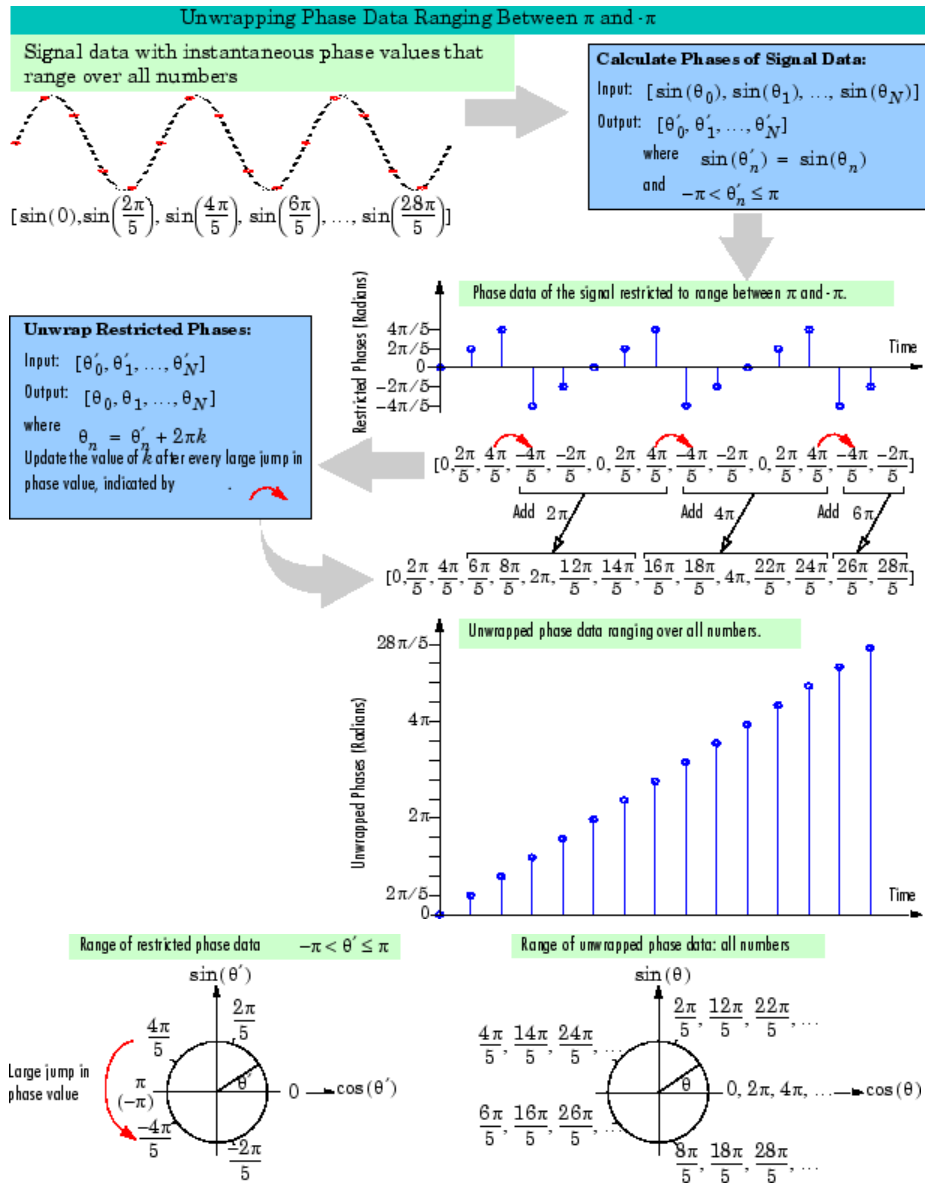
**Dialog
Box**
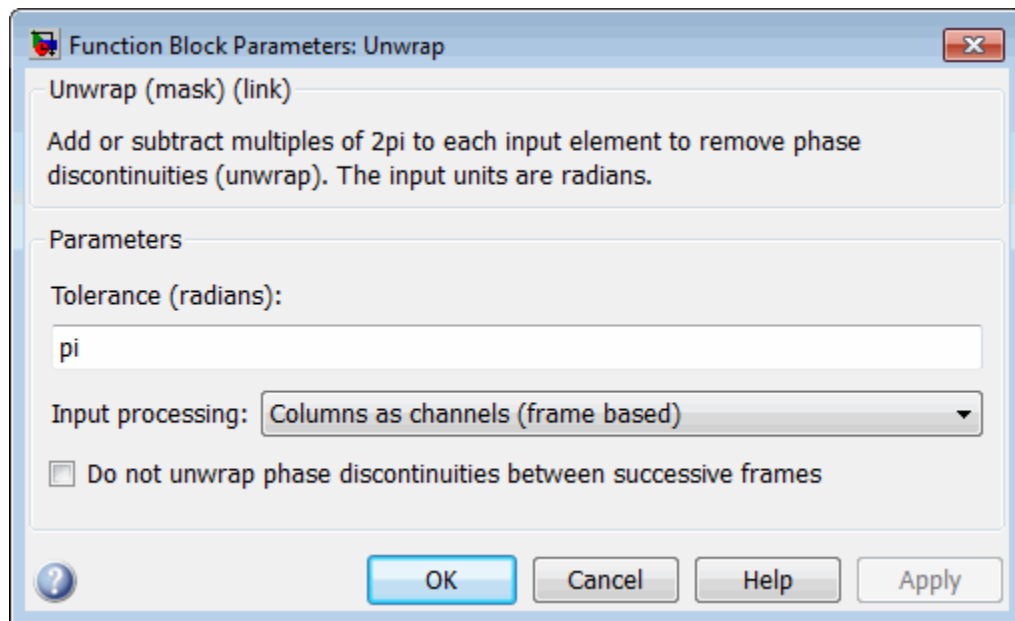
Tolerance
>   The jump size that the block recognizes as a true phase
>   discontinuity. The default is set to π (rather than a smaller value)
>   to avoid altering legitimate signal features. To increase the block's
>   sensitivity, set the **Tolerance** to a value slightly less than π.

Input processing
>   Specify how the block should process the input. You can set this
>   parameter to one of the following options:
>
>   - Columns as channels (frame based) — When you select
>   this option, the block treats each column of the input as a
>   separate channel.
>
>   - Elements as channels (sample based) — When you select
>   this option, the block treats each element of the input as a
>   separate channel.

Unwrap

> **Note** The Inherited (`this choice will be removed - see release notes`) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Do not unwrap phase discontinuities between successive frames
When you clear this check box, the block ignores boundaries between input frames and does not reset the initial phase value to zero each time a new input is received. In this mode, the block continuously unwraps the data in each column of the input. When you select this check box, the block treats each frame of input data independently, and resets the initial phase value for each new input frame. See the "Frame-Based Processing" on page 2-1630 section for more information.

This parameter is available only when you configure the block to perform frame-based processing. In sample-based processing mode, the block does not reset the initial phase value to zero for each new input. See "Sample-Based Processing" on page 2-1632 for more information.

Supported Data Types
- Double-precision floating point
- Single-precision floating point

See Also   unwrap   MATLAB

Purpose   Resample input at higher rate by inserting zeros

Library   Signal Operations

                     dspsigops

Description   The Upsample block resamples each channel of the M_i-by-N input at
                     a rate L times higher than the input sample rate by inserting L-1
                     zeros between consecutive samples. You specify the integer L in the
                     Upsample factor parameter. The **Sample offset** parameter, D,
                     allows you to delay the output samples by an integer number of sample
                     periods. Doing so enables you to select any of the L possible output
                     phases. The value you specify for the **Sample offset** parameter must

                     be in the range $0 \le D < (L-1)$.

                     You can use this block inside of triggered subsystems when you set the
                     Rate options parameter to Enforce single-rate processing.

                     Frame-Based Processing

                     When you set the **Input processing** parameter to Columns as
                     channels (frame based), the block upsamples each column of the
                     input over time. In this mode, the block can perform either single-rate
                     or multirate processing. You can use the **Rate options** parameter to
                     specify how the block upsamples the input:

                     • When you set the **Rate options** parameter to Enforce single-rate
                       processing, the input and output of the block have the same sample
                       rate. In this mode, the block outputs a signal with 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.

                       For an example of single-rate upsampling, see Example: Single-Rate
                       Processing on page 2-1639.

                     • When you set the **Rate options** parameter to Allow multirate
                       processing, the block treats an M_i-by-N matrix input as N
                       independent channels. The block upsamples each column of the input

Upsample

over time by keeping the frame size constant ($M_i = M_o$), and making the output frame period (T_{fo}) L times shorter than the input frame period ($T_{fo} = T_{fi}/L$).

See Example: Multirate, Frame-Based Processing on page 2-1639 for an example that uses the Upsample block in this mode.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-N matrix input as $M*N$ independent channels, and upsamples each channel over time. In this mode, the block always performs multirate processing. 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.

Zero Latency

The Upsample block has *zero-tasking latency* for all single-rate operations. The block is in a single-rate mode if you set the **Upsample factor** parameter to `1` or if you set the **Input processing** parameter to `Columns as channels (frame based)` and the **Rate options** parameter to `Enforce single-rate processing`.

The Upsample block also has zero-tasking latency for multirate operations if you run your model in Simulink single-tasking mode.

Zero-tasking latency means that the block propagates the first input (received at *t*=0) immediately following the *D* consecutive zeros specified by the **Sample offset** parameter. This output (*D*+1) is followed in turn by the *L*-1 inserted zeros and the next input sample.

Nonzero Latency

The Upsample block has tasking latency for multirate, multitasking operation:

- In multirate, sample-based processing 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 conditions** 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 multirate, frame-based processing 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 conditions** parameter can be an M_i-by-N matrix, or a scalar to be repeated across all elements of the input matrix.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and "Scheduling" in the *Simulink Coder User's Guide*.

Examples

Example: Single-Rate Processing

In the ex_upsample_ref2 model, the Upsample block resamples a single-channel input with a frame size of 16. The block upsamples the input by a factor of 4. Thus, the output of the block has a frame size of 64. Because the block is in single-rate processing mode, the input and output frame rates are identical.
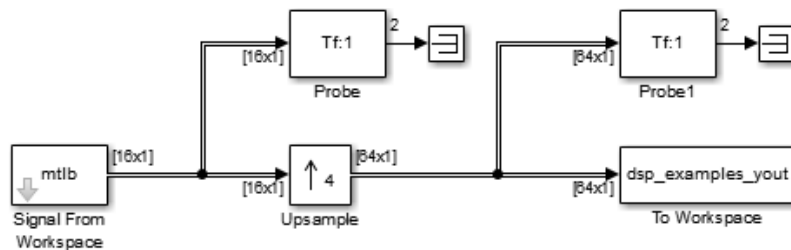
Example: Multirate, Frame-Based Processing

In the ex_upsample_ref1 model, the Upsample block resamples a single-channel input with a frame period of 1 second. The block

Upsample

upsamples the input by a factor of 4. Thus, the output of the block has a frame period of 0.25 seconds. Because the block is in multirate processing mode, the input and output frame sizes are identical.
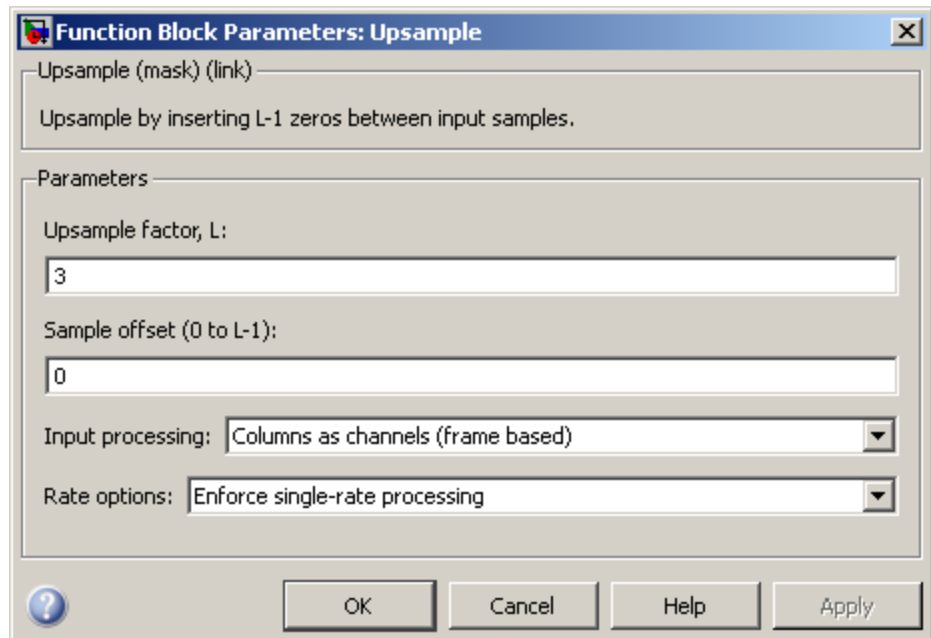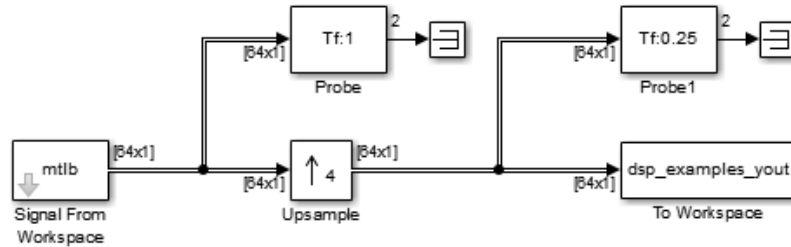
Dialog Box

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].

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel. In this mode, the block can perform single-rate or multirate processing.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel. In this mode, the block always performs multirate processing.

Note The option `Inherit from input (this choice will be removed - see release notes)` will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Rate options

Specify the method by which the block upsamples the input. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate by increasing the output frame size by a factor of *L*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.

- `Allow multirate processing` — When you select this option, the block resamples the signal such that the output sample rate is *L* times faster than the input sample rate.

Upsample

Initial conditions

The value with which the block is initialized for cases of nonzero latency, a scalar or matrix. This value appears in the output as sample $D+1$. This parameter appears only when you configure the block to perform multirate processing.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Downsample	DSP System Toolbox
FIR Interpolation	DSP System Toolbox
FIR Rate Conversion	DSP System Toolbox
Repeat	DSP System Toolbox

Purpose   Delay input by time-varying fractional number of sample periods

Library   Signal Operations

dspsigops

Description   The Variable Fractional Delay block delays each element of the
discrete-time N-D input array, *u*, by a variable number of sample
intervals. The input delay values can be integer or noninteger values.
The block provides three different interpolation modes: Linear, FIR,
and Farrow.

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 **Interpolation mode**
parameter.

- In Linear interpolation mode, the block stores the D_{max}+1 most
  recent samples received at the In port for each channel, where D_{max} is
  the value you specify for the **Maximum delay (Dmax) in samples**
  parameter.

- In FIR interpolation mode, the block stores the D_{max}+P+1 most recent
  samples received at the In port for each channel, where P is the value
  you specify for the **Interpolation filter half-length (P)** parameter.

- In Farrow interpolation mode, the block stores the $D_{max}+\dfrac{N}{2}+1$ most
  recent samples received at the In port for each channel, where N is
  the value you specify for the **Farrow filter length (N)** parameter.

The Variable Fractional Delay block assumes that the input values
at the Delay port are between D_{min} and D_{max}, where D_{min} appears in
the Valid delay range section on the **Main** pane of the block mask,
and D_{max} is the value of the **Maximum delay (Dmax) in samples**
parameter. The block clips delay values less than D_{min} to D_{min} and delay
values greater than D_{max} to D_{max}.

2-1643

Variable Fractional Delay

You must consider additional factors when selecting valid `Delay` values for the `FIR` and `Farrow` interpolation modes. For more information about these considerations, refer to FIR Interpolation Mode on page 1646 or Farrow Interpolation Mode on page 1648, respectively.

The Variable Fractional Delay block is similar to the Variable Integer Delay block, in that they both store a minimum of D_{max}+1 past samples in memory. The Variable Fractional Delay block differs only in the way that these stored samples are accessed; a fractional delay requires the computation of a value by interpolation from the nearby samples in memory.

Sample-Based Processing

When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element of the N-D input array, u, as an independent channel. The input to the Delay port, v, must either be an N-D array of the same size and dimension as the input u, or be a scalar value, such that $D_{min} \leq v \leq D_{max}$.

For example, consider an M-by-N input matrix. The block treats each of the $M*N$ matrix elements as independent channels. The input to the Delay port can be an M-by-N matrix of floating-point values in the range $D_{min} \leq v \leq D_{max}$ that specifies the number of sample intervals to delay each channel of the input, or it can be a scalar floating-point value, $D_{min} \leq v \leq D_{max}$, by which to equally delay all channels.

In sample-based processing mode, the block treats an unoriented vector input as an M-by-1 matrix. In this mode, the output is also an unoriented vector.

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation in the same manner as the Variable Integer Delay block. See the Variable Integer Delay block reference page for more information.

Frame-Based Processing

When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each of the N input columns as an independent channel containing M_i sequential time samples.

The input to the Delay port, v, contains floating-point values that specify the number of sample intervals to delay the current input.

The input to the Delay port can be a scalar value to uniformly delay every sample in every channel. It can also be a length-M column vector, containing one delay for each sample in the input frame. The block applies the set of delays contained in the vector identically to every channel of a multichannel input. The Delay port entry can also be a length-N row vector, containing one delay for each channel. Finally, the Delay port entry can be an M-by-N matrix, containing a different delay for each corresponding element of the 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 block applies the set of fractional delays contained in v identically to every channel of a multichannel input.

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation in the same manner as the Variable Integer Delay block. See the Variable Integer Delay block reference page for more information.

Interpolation Modes

The delay value specified at the Delay port serves as an index into the block's memory, `U`, which stores, at a minimum, the $D_{max}+1$ 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)`. The block computes fractional delays by interpolating between stored samples; the three available interpolation modes are `Linear`, `FIR` and `Farrow`.

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.

Variable Fractional Delay

```
vi = floor(v)      % vi = integer delay
vf = v-vi      % vf = fractional delay
y = (1-vf)*U(vi+1) + vf*U(vi)
```

FIR Interpolation Mode

In `FIR Interpolation` mode, the block provides a discrete set of fractional delays described by:

$$v + \frac{i}{L}, \qquad v \geq P - 1, \ \ i = 0, 1, ..., L - 1$$

If v is less than P-1, the block's behavior depends on the setting of the **For small input delay values** parameter. You can specify the block's behavior when the input delay value is too small to center the kernel (less than P-1), by setting the **For small input delay values** parameter:

- If you select `Clip to the minimum value necessary for centered kernel`, the block remains in `FIR` interpolation mode by clipping small input delay values to the smallest value necessary to center the kernel.

  To determine the minimum delay value, select `Clip to the minimum value necessary for centered kernel`, and click **Apply** on the block mask. All input delay values less than the value displayed for D_{min} will be clipped to D_{min}.

- If you select `Switch to linear interpolation if kernel cannot be centered`, the block computes fractional delays using linear interpolation when the input delay value is less than P-1.

To add an extra delay to the minimum possible delay value, select the **Disable direct feedthrough by increasing minimum possible delay by one** check box. Checking this box prevents algebraic loops from occurring when you use the block inside a feedback loop.

In `FIR Interpolation` mode, the block implements a polyphase structure to compute a value for each sample at the desired delay. Each arm of the structure corresponds to a different delay value and the

output computed for each sample corresponds to the output of the arm with a delay value nearest to the desired input delay. Thus, only a discrete set of delays is actually possible. The number of coefficients in each of the *L* filter arms of the polyphase structure is 2*P*. In most cases, using values of *P* between 4 and 6 will provide you with reasonably accurate interpolation values.

In this mode, the Signal Processing Toolbox `intfilt` function computes an FIR filter for interpolation.

For example, when you set the parameters on the block mask to the following values:

- Interpolation filter half-length (P): 4

- Interpolation points per input sample: 10

- Normalized input bandwidth: 1

The filter coefficients are given by:

```
b = intfilt(10,4,1);
```

The block then implements this filter as a polyphase structure, as described previously.

Increasing the **Interpolation filter half length (P)** increases the accuracy of the interpolation, but also 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** (*L*) increases the number of representable discrete delay points, but also increases the simulation's memory requirements and does not affect the computational load per sample.

The **Normalized input bandwidth (0 to 1)** 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 (0 to 1) to constrain the frequency content of the output to that range.

Note You can consider each of the *L* interpolation filters to correspond to one output phase of an "upsample-by-*L*" FIR filter. Thus, the **Normalized input bandwidth (0 to 1)** value improves the stopband in critical regions, and relaxes the stopband requirements in frequency regions where there is no signal energy.

Farrow Interpolation Mode

In `Farrow` interpolation mode, the block uses the LaGrange method to interpolate values.

To increase the minimum possible delay value, select the **Disable direct feedthrough by increasing minimum possible delay by one** check box. Checking this box prevents algebraic loops from occurring when you use the block inside a feedback loop.

To specify the block's behavior when the input delay value is too small to center the kernel (less than $\frac{N}{2}$ -1), set the **For small input delay values** parameter:

- If you select `Clip to the minimum value necessary for centered kernel`, the block clips small input delay values to the smallest value necessary to keep the kernel centered. This increases D_{min} but yields more accurate interpolation values.

  To determine the minimum delay value, select `Clip to the minimum value necessary for centered kernel`, and click **Apply** on the block mask. All input delay values less than the value displayed for D_{min} will be clipped to D_{min}.

- If you select `Use off-centered kernel`, the block computes fractional delays using a Farrow filter with an off-centered kernel. This mode does not increase D_{min}, but if there are input delay values

less than $\dfrac{N}{2}$ -1, the results are less accurate than the results achieved by keeping the kernel centered.

Fixed-Point Data Types

The diagrams in the following sections show the data types used within the Variable Fractional Delay block for fixed-point signals.

Although you can specify most of these data types on the **Data Types** pane of the block mask, the following data types are computed internally by the block and cannot be directly specified on the block mask.

Data Type	Word Length	Fraction Length
vf data type	Same word length as the **Coefficients**	Same as the word length
HoldInteger data type	Same word length as the input delay value	0 bits
Integer data type	32 bits	0 bits

Note When the block input is fixed point, all internal data types are signed fixed point.

To compute the integer (v_i) and fractional (v_f) parts of the input delay value (v), the Variable Fractional Delay block uses the following equations:

Variable Fractional Delay

$$D_{min} < v < D_{max} \Rightarrow \begin{cases} v_i = \text{floor}(v) \\ v_f = v - vi \end{cases}$$

$$v \leq D_{min} \Rightarrow \begin{cases} v_i = D_{min} \\ v_f = 0 \end{cases}$$

$$v \geq D_{max} \Rightarrow \begin{cases} v_i = D_{max} \\ v_f = 0 \end{cases}$$

Linear Interpolation Mode

The following diagram shows the fixed-point data types used by the Linear interpolation mode of the Variable Fractional Delay block.
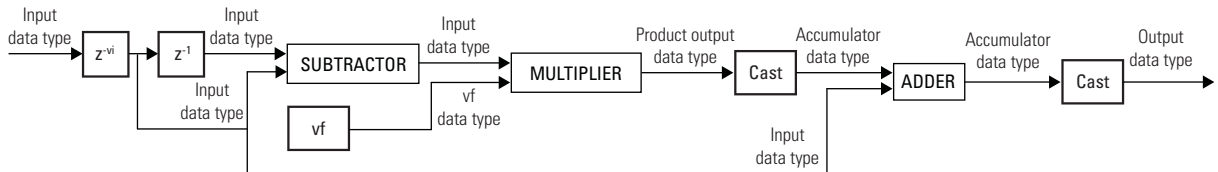
FIR Interpolation Mode

The following diagram illustrates how the Variable Fractional Delay block selects the arm of the polyphase filter structure that most closely matches the fractional delay value (v_f).
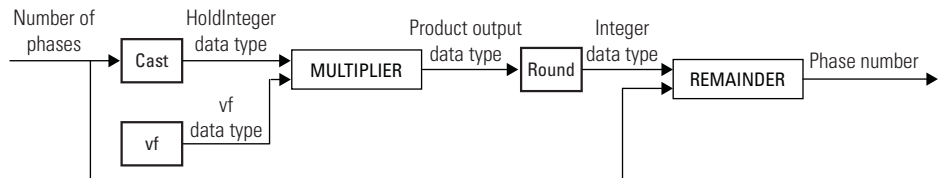
See the "Fixed-Point Data Types" on page 2-633 section of the FIR Interpolation reference page for a diagram showing the fixed-point data types used by the Variable Fractional Delay block in FIR interpolation mode.

Farrow Interpolation Mode

The following diagram shows the fixed-point data types used by the Farrow interpolation mode of the Variable Fractional Delay block where:

- **Farrow filter length (N)** = 4

- **For small input delay values** = Clip to the minimum value necessary to for centered kernel
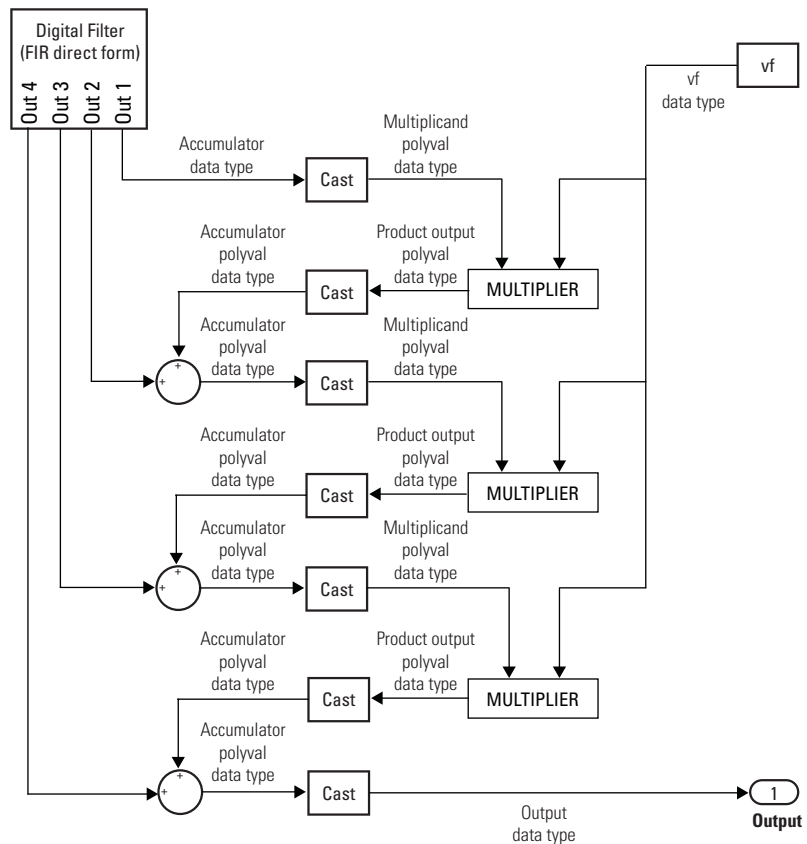
Variable Fractional Delay

The following diagram shows the fixed-point data types used by the Farrow interpolation mode of the Variable Fractional Delay block where:

- **Farrow filter length (N)** = 4

- **For small input delay values** = Use off-centered kernel
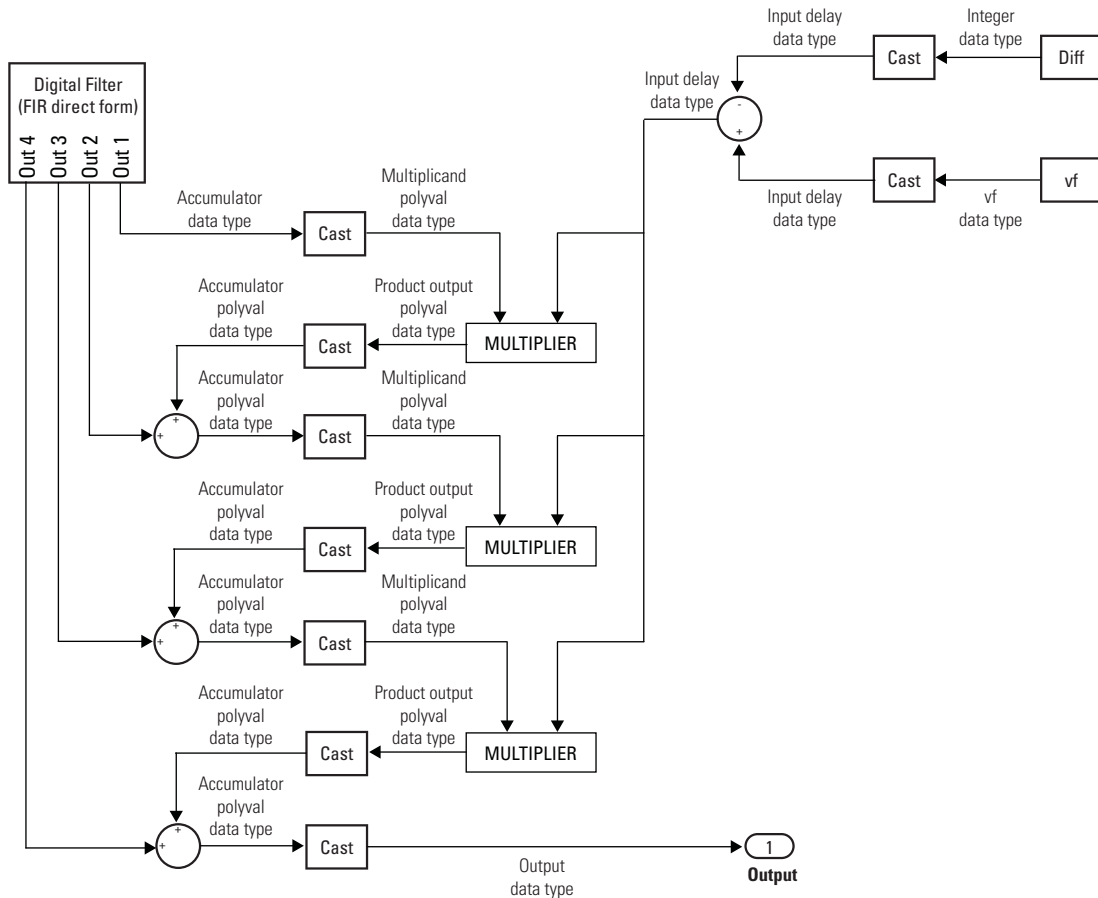
Diff is computed from the integer part of the delay value (v_i) and the **Farrow filter length (N)** according to the following equation:

$$Diff = v_i - \left(\frac{N-1}{2}\right)$$
$$Diff \geq 0 \Rightarrow Diff = 0$$
$$Diff < 0 \Rightarrow Diff = -Diff$$

The following diagram shows the fixed-point data types used by the Digital Filter block's FIR direct form filter.
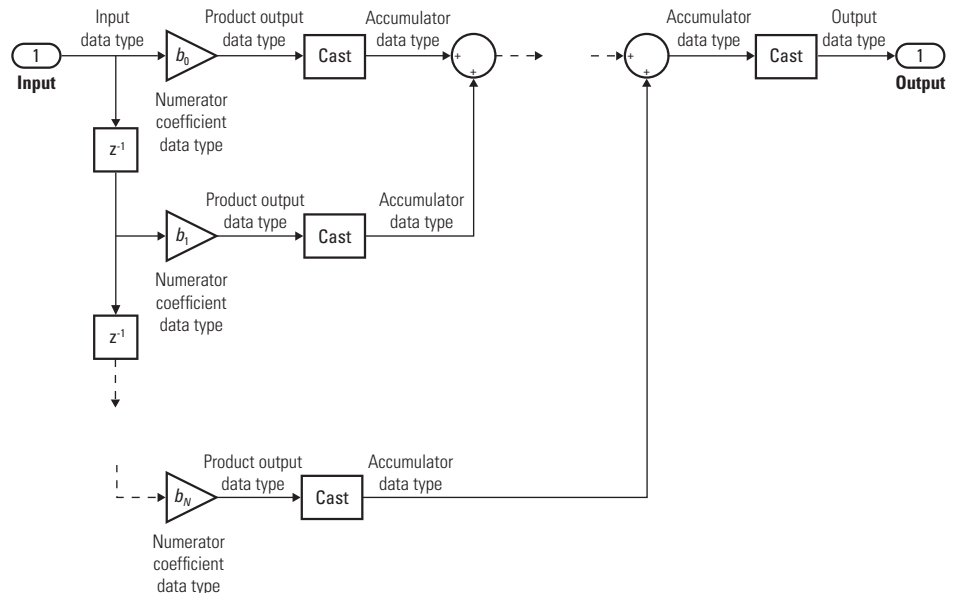
Examples

The dspaudioeffects example illustrates three audio effects applied to a short segment of music. When you set the **Audio effect** of the Effect block to Flanging, the model uses the Variable Fractional Delay block to mix the original signal with a delayed version of itself.

To see the Flanging subsystem, right-click the Effect block, and select **Diagram > Mask > Look Under Mask**. Next, double-click the Flanging block in the Effect block subsystem that just opened. The

Variable Fractional Delay

Flanging subsystem opens, and you can see the parameters of the Variable Fractional Delay block.

Dialog Box

The **Main** pane of the Variable Fractional Delay block dialog appears as follows.

Function Block Parameters: Variable Fractional Delay

Variable Fractional Delay

Delay discrete-time input by the time-varying fractional number of sample periods specified by the 'Delay' input. The block provides Linear, FIR, and Farrow interpolation modes. In FIR mode, the filter is designed using the 'intfilt' function from the Signal Processing Toolbox.

The input delay is clipped to a valid range (Dmin to Dmax) that is determined by the parameter settings.

| Main | Data Types |

General parameters

Interpolation mode: Linear

Maximum delay (Dmax) in samples: 100

Input processing: Columns as channels (frame based)

Initial conditions: 0

☐ Disable direct feedthrough by increasing minimum possible delay by one

Valid delay range (in samples)

Dmin: 0

Dmax: 100

OK   Cancel   Help   Apply

Variable Fractional Delay

Interpolation 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.

Interpolation filter half-length (P)

Half the number of input samples to use in the FIR interpolation filter. This parameter is only visible when the **Interpolation mode** is set to FIR.

Farrow filter length (N)

The number of input samples to use in the Farrow interpolation filter. This parameter is only visible when the **Interpolation mode** is set to Farrow.

Interpolation points per input sample

The number of points per input sample, L, at which a unique FIR interpolation filter is computed. This parameter is only visible when the **Interpolation mode** is set to FIR.

Normalized input bandwidth (0 to 1)

The bandwidth to which the interpolated output samples should be constrained. The value must be a real scalar between 0 and 1. A value of 1 specifies half the sample frequency. This parameter is only visible when the **Interpolation mode** is set to FIR.

Maximum delay (Dmax) in samples

The maximum delay that the block can produce, D_{max}. Input delay values exceeding this maximum are clipped to D_{max}.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

> **Note** The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Initial conditions

The values with which the block's memory is initialized. See the Variable Integer Delay block for more information.

Disable direct feedthrough by increasing minimum possible delay by one

Select this box to disable direct feedthrough by increasing the minimum possible delay value. When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block increases the minimum possible delay value by *frame size* − 1. Similarly, when you set the **Input processing** parameter to `Elements as channels (sample based)`, the block increases the minimum possible delay value by one sample.

Checking this box allows you to use the Variable Fractional Delay block in feedback loops.

For small input delay values

Specify the block's behavior when the input delay values are too small to center the kernel. This parameter is only visible when the **Interpolation mode** is set to `FIR` or `Farrow`.

You can specify how the block handles input delay values that are too small for the kernel to be centered using one of the following choices:

- In both `FIR` and `Farrow` interpolation modes, you can select `Clip to the minimum value necessary for centered kernel`. This option forces the block to increase D_{min} to the smallest value necessary to keep the kernel centered.

Variable Fractional Delay

- In `FIR` interpolation mode, you can select `Switch to linear interpolation if kernel cannot be centered`. This option forces the block to preserve the value of D_{min} and compute all interpolated values using `Linear` interpolation.

- In `Farrow` interpolation mode, you can select `Use off-centered kernel`. This option forces the block to preserve the value of D_{min} and compute the interpolated values using a farrow filter with an off-centered kernel.

Valid delay range

The values displayed in this section of the **Main** pane are calculated (in samples) by the block based on the current parameter settings.

- `Dmin` is the smallest possible valid delay value (in samples) based on the current settings of the block parameters. The block clips all input delay values less than `Dmin` to `Dmin`.

- `Dmax` is the maximum valid delay value (in samples) based on the current settings of the block parameters. The block clips all input delay values greater than `Dmax` to `Dmax`.

The **Data Types** pane of the Variable Fractional Delay block dialog appears as follows.

Function Block Parameters: Variable Fractional Delay

Variable Fractional Delay

Delay discrete-time input by the time-varying fractional number of sample periods specified by the 'Delay' input. The block provides Linear, FIR, and Farrow interpolation modes. In FIR mode, the filter is designed using the 'intfilt' function from the Signal Processing Toolbox.

The input delay is clipped to a valid range (Dmin to Dmax) that is determined by the parameter settings.

| Main | Data Types |

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input. When the block input is fixed point, all internal data types are signed fixed point.

Fixed-point operational parameters

Rounding mode: Zero   Overflow mode: Wrap

Fixed-point data types

Data Type

Coefficients   Same word length as input

Product output   Same as first input

Accumulator   Same as product output

Output   Same as accumulator

☐ Lock data type settings against changes by the fixed-point tools

OK   Cancel   Help   Apply

Variable Fractional Delay

Rounding mode
: Select the rounding mode for fixed-point operations.

Overflow mode
: Select the overflow mode for fixed-point operations.

Coefficients
: Choose how you specify the word length and fraction length of the filter coefficients.

- When you select `Same word length as input`, the word length of the filter coefficients match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Specify word length`, you can enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

Product output
: Use this parameter to specify how you would like to designate the product output word and fraction lengths. See "Fixed-Point Data Types" on page 2-1649 and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block.

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

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

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

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See "Fixed-Point Data Types" on page 2-1649 and "Multiplication Data Types" for illustrations depicting the use of the accumulator data type in this block:

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

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

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

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

Product output polyval

Choose how you specify the word length and fraction length of the product output polyval data type. This parameter is only visible when the **Interpolation mode** is set to `Farrow`.

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

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

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

Accumulator polyval

Choose how you specify the word length and fraction length of the accumulator polyval data type. This parameter is only visible when the **Interpolation mode** is set to `Farrow`.

Variable Fractional Delay

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

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

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

Multiplicand polyval
Choose how you specify the word length and fraction length of the multiplicand polyval data type. This parameter is only visible when the **Interpolation mode** is set to `Farrow`.

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

- When you select `Binary point scaling`, you can enter the word length and the fraction length of the multiplicand polyval, in bits.

- When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the multiplicand polyval. This block requires power-of-two slope and a bias of zero.

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

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

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

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

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

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Delay	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Variable Fractional Delay

See Also

Delay	DSP System Toolbox
Unit Delay	Simulink
Variable Integer Delay	DSP System Toolbox

Purpose   Delay input by time-varying integer number of sample periods

Library   Signal Operations

dspsigops

Description
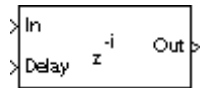
The Variable Integer Delay block delays the discrete-time input at the In port by the integer number of sample intervals specified by the input to the Delay port. The sample rate of the input signal at the Delay port must be the same as the sample rate of the input signal at the In port. When these sample rates are not the same, you need to insert a Zero-Order Hold or Rate Transition block in order to make the sample rates identical. When you set the **Input processing** parameter to Elements as channels (sample based), the delay for an N-D input can be a scalar value to uniformly delay every sample in every channel, or a matrix containing one delay value for each channel of the input. When you set the **Input processing** parameter to Columns as channels (frame based), the delay can be a scalar value to uniformly delay every sample in every channel, a vector containing one delay value for each sample in the input frame, or a vector containing one delay value for each channel in the input frame.

The delay values should be in the range of 0 to D, where D is the **Maximum delay**. Delay values greater than D or less than 0 are clipped to those respective values and noninteger delays are rounded to the nearest integer value.

The Variable Integer Delay block differs from the Delay block in the following ways.

Variable Integer Delay

Variable Integer Delay Block	Delay Block
The delay is provided as an input to the Delay port.	You specify the delay as a parameter setting in the dialog box.
Delay can vary with time; for example, when the block performs frame-based processing, the nth element's delay in the first input frame can differ from the nth element's delay in the second input frame.	Delay cannot vary with time; for example, when the block performs frame-based processing, the nth element's delay is the same for every input frame.
When you use the Variable Integer Delay block in a feedback loop, you must check the **Disable direct feedthrough by increasing minimum possible delay by one** check box. This prevents the occurrence of an algebraic loop when the delay of the Variable Integer Delay block is driven to zero.	You can use the Delay block to break an algebraic loop.

Sample-Based Processing

When you set the **Input processing** parameter to Elements as channels (sample based), the Variable Integer Delay block supports N-D input arrays. When the input is an M-by-N-by-P array, the block treats each of the $M*N*P$ elements as independent channels, and applies the delay at the Delay port to each channel.

The Variable Integer Delay block stores the D+1 most recent samples received at the In port for each channel. At each sample time the block outputs the stored sample(s) indexed by the input to the Delay port.

For example, when the input to the In port, u, is a scalar signal, the block stores a vector, U, of the D+1 most recent signal samples. When the current input sample is U(1), the previous input sample is U(2), and so on, then the block's output is

```
y = U(v+1);      % Equivalent MATLAB code
```

where v is the input to the Delay port. 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 next figure 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.
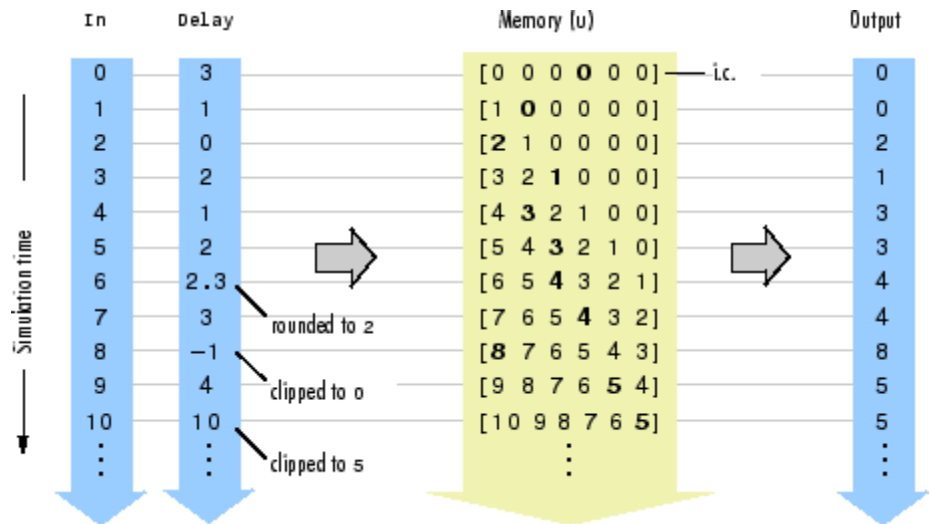
The current input at each time step is immediately stored in memory as U(1). This allows the current input to be available at the output for a delay of 0 (v=0).

Variable Integer Delay

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Unlike the Delay block, the Variable Integer Delay block does not have a fixed initial delay period during which the initial conditions appear at the output. Instead, the initial conditions are propagated to the output only when they are indexed in memory by the value at the Delay port. Both fixed and time-varying initial conditions can be specified in a variety of ways to suit the dimensions of the input sequence.

Fixed Initial Conditions

The settings in this section 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 in this figure,

  Maximum delay (samples):

  100

  Initial conditions:

  0

  the block initializes 100 M-by-N matrices in memory with zeros.

- Array of size M-by-N-by-D. In this case, you can specify different fixed initial conditions for each channel. See the Array bullet in "Time-Varying Initial Conditions" on page 2-1668 below for details.

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 in the next figure, the block initializes U(2:6) with values [-1, -1, -1, 0, 1].

Maximum delay (samples):
5
Initial conditions:
[-1 -1 -1 0 1]

- Array of dimension *M*-by-*N*-by-*D* with which to initialize memory samples U(2:D+1), where *D* is the **Maximum delay** and *M* and *N* are the number of rows and columns, respectively, in the input matrix. For a 2-by-3 input and the following parameters, the block initializes memory locations U(2:5) with values

$$\mathbf{U}(2) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \ \mathbf{U}(3) = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}, \ \mathbf{U}(4) = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix}, \ \mathbf{U}(5) = \begin{bmatrix} 4 & 4 & 4 \\ 4 & 4 & 4 \end{bmatrix}$$

Maximum delay (samples):
4
Initial conditions:
cat(3, [1 1 1; 1 1 1], [2 2 2; 2 2 2], [3 3 3; 3 3 3], [4 4 4; 4 4 4])

An *M*-by-*N*-by-*P*-by-*D* array can be entered for the **Initial Conditions** parameter when the input is an *M*-by-*N*-by-*P* array. The (*M,N,P,T*)th sample of the **Initial Conditions** matrix provides the initial condition value for the (*M,N,P*)th channel of the input matrix at delay = *D*–*t*+1 samples.

Frame-Based Processing

When you set the **Input processing** parameter to Columns as channels (frame based), the input can be an *M*-by-*N* matrix. The block treats each of the *N* input columns as independent channels containing *M* sequential time samples.

Variable Integer Delay

In this 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 column vector containing one delay value for each sample in the input frame(s). The set of delays contained in the vector is applied identically to every channel of a multichannel input. The Delay port entry can also be a length-N row vector, containing one delay for each channel. Finally, the Delay port entry can be an M-by-N matrix, containing a different delay for each corresponding element of the input.

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. 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)
```
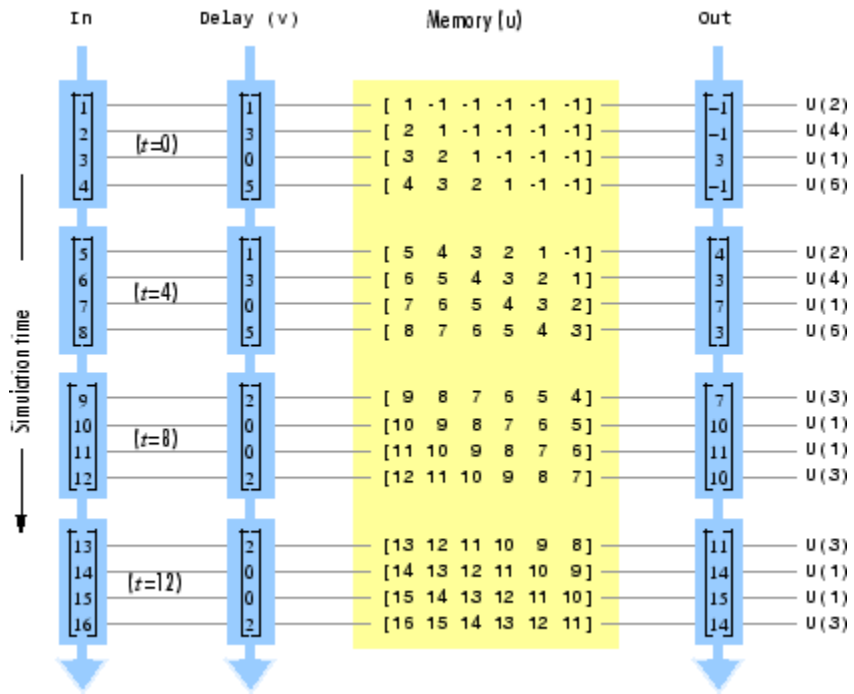
The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Both fixed and time-varying initial conditions can be specified.

Fixed Initial Conditions

The settings shown in this section 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.

Variable Integer Delay

- Array of size 1-by-*N*-by-*D*. In this case, you can specify different fixed initial conditions for each channel. See the Array bullet in "Time-Varying Initial Conditions" on page 2-1672 below for details.

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. When the block is performing frame-based processing, you can specify a time-varying initial condition in the following ways:

- Vector containing *D* elements. 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 [1:100; 1:100]' with a frame size of 4, delay of 5, and the following parameter settings, the block outputs the following sequence of frames at the start of the simulation:

$$
\begin{bmatrix} -1 & -1 \\ -2 & -2 \\ -3 & -3 \\ -4 & -4 \end{bmatrix}, \begin{bmatrix} -5 & -5 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \cdots
$$

- 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

[1:100; 1:100]' with a frame size of 4, delay of 5, and the following parameter settings, the block outputs the following sequence of frames at the start of the simulation:

$$\begin{bmatrix} -1 & -11 \\ -2 & -22 \\ -3 & -33 \\ -4 & -44 \end{bmatrix}, \begin{bmatrix} -5 & -55 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \dots$$

Maximum delay (samples):

5

Initial conditions:

cat(3, [-1 -11], [-2 -22], [-3 -33], [-4 -44], [-5 -55])
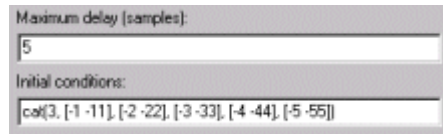
By specifying a 1-by-N-by-D initial condition array such that each 1-by-N vector entry is identical, you can implement different fixed initial conditions for each channel.

Examples   See "Basic Algorithmic Delay" in the *DSP System Toolbox User's Guide*.

Variable Integer Delay
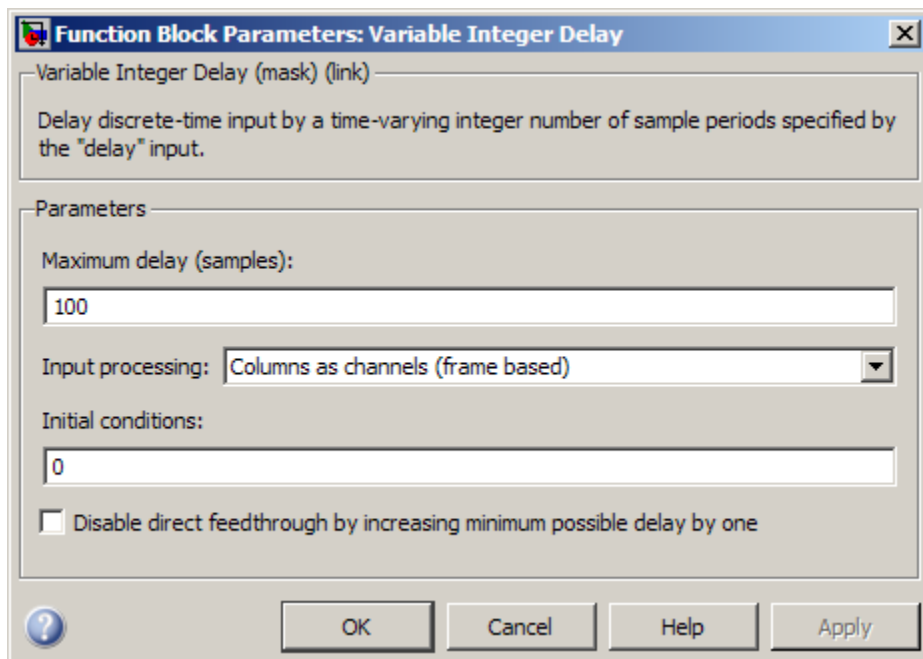
Dialog Box

Maximum delay

The maximum delay that the block can produce for any sample. Delay input values exceeding this maximum are clipped at the maximum.

Initial conditions

The values with which the block's memory is initialized.

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

> **Note** The `Inherited (this choice will be removed - see release notes)` option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Disable direct feedthrough by increasing minimum possible delay by one

Select this box to disable direct feedthrough by adding one to the minimum possible delay value. When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block increases the minimum possible delay value by *frame size* -1. Similarly, when you set the **Input processing** parameter to `Elements as channels (sample based)`, the block increases the minimum possible delay value by one sample.

Checking this box allows you to use the Variable Integer Delay block in feedback loops.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers

Variable Integer Delay

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers
Delay	• Double-precision floating point
	• Single-precision floating point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Delay	DSP System Toolbox
Variable Fractional Delay	DSP System Toolbox

Purpose   Select subset of rows or columns from input

Library   Signal Management / Indexing

dspindex
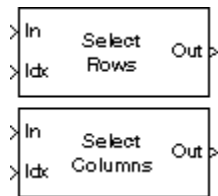
Description   The Variable Selector block extracts a subset of rows or columns from
the *M*-by-*N* input matrix u at each input port. You specify the number
of input and output ports in the **Number of input signals** parameter.

When the **Select** parameter is set to Rows, the Variable Selector block
extracts rows from each input matrix, while if the **Select** parameter
is set to Columns, the block extracts columns.

When the **Selector mode** parameter is set to Variable, the length-*L*
vector input to the Idx port selects *L* rows or columns of each input to
pass through to the output. The elements of the indexing vector can be
updated at each sample time, but the vector length must remain the
same throughout the simulation.

When the **Selector mode** parameter is set to Fixed, the Idx port is
disabled, and the length-*L* vector specified in the **Elements** parameter
selects *L* rows or columns of each input to pass through to the output.
The **Elements** parameter is tunable, so you can change the values
of the indexing vector elements at any time during the simulation;
however, the vector length must remain the same.

For both variable and fixed indexing modes, the row selection operation
is equivalent to

```
y = u(idx,:)      % Equivalent MATLAB code
```

and the column selection operation is equivalent to

```
y = u(:,idx)      % Equivalent MATLAB code
```

where idx is the length-*L* indexing vector. The row selection output size
is *L*-by-*N* and the column selection output size is *M*-by-*L*. Input rows or
columns can appear any number of times in the output, or not at all.

Variable Selector

When the input is an unoriented vector, the **Select** parameter is ignored; the output is a unoriented 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 action you specify using the **Invalid index** parameter.

When the indexing vector elements are of Boolean data type, the block performs logical indexing. Select **Fill empty spaces in outputs (for logical indexing)** to access the **Fill values** parameter. These values are appended to the output to make it as long as the input elements.

Note The Variable Selector block always copies the selected input rows to a contiguous block of memory (unlike the Simulink Selector block).

Function Block Parameters: Variable Selector

Variable Selector (mask) (link)

Selects and/or reorders the rows or columns of the input according to a specified vector of indices (the indices need not be unique). The 'Selector mode' parameter determines whether the block uses the same indices for every input ('Fixed'), or uses different indices for every input ('Variable'). When set to 'Variable', you provide the vector of indices through an input port.

When the indices are of Boolean data type the block performs logical indexing.

Parameters

Number of input signals:

1

Select: Rows

Selector mode: Variable

Index mode: One-based

Invalid index: Clip Index

☑ Fill empty spaces in outputs (for logical indexing)

Fill values:

0

OK   Cancel   Help   Apply

Dialog

Variable Selector

Box

Number of input signals

Specify the number of input signals. An input port is created on the block for each input signal.

Select

Specify the dimension of the input to select, `Rows` or `Columns`.

Selector mode

Specify 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

Specify a vector containing the indices of the input rows or columns that will appear in the output matrix. This parameter appears only when you set the **Selector mode** to `Fixed`.

Index mode

When set to `One-based`, an index value of 1 refers to the first row or column of the input. When set to `Zero-based`, an index value of 0 refers to the first row or column of the input.

Invalid index

Specify how the block handles an invalid index value. You can select one of the following options:

- `Clip index` — Clip the index to the nearest valid value, and do not issue an alert.

  For example, if the block receives a `64`-by-`4` input and the **Select** parameter is set to `Rows`, the block clips an index of `72` to `64`. For the same input, if the **Select** parameter is set to `Columns`, the block clips an index of `72` to `4`. In both cases, the block clips an index of `-2` to `1`.

- `Clip and warn` — Clip the index to the nearest valid value and display a warning message at the MATLAB command line.

- Generate error — Display an error dialog box and terminate the simulation.

This parameter is tunable.

Fill empty spaces in outputs (for logical indexing)
When the indexing vector elements are of Boolean data type, the block performs logical indexing. This can cause empty spaces in the output. Select this parameter to designate values to be appended to the output in the **Fill values** parameter.

Fill values
Specify the fill values when the block performs logical indexing. This parameter appears only when you select the **Fill empty spaces in outputs (for logical indexing)** check box.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	• Enumerated
Idx	• Double-precision floating point
	• Single-precision floating point
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Variable Selector

Port	Supported Data Types
	• Enumerated
Out	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	• Enumerated

See Also

Multiport Selector	DSP System Toolbox
Permute Matrix	DSP System Toolbox
Selector	Simulink
Submatrix	DSP System Toolbox

Purpose   Compute variance of input or sequence of inputs

Library   Statistics

dspstat3

Description The Variance block computes the unbiased variance of each row or
column of the input, along vectors of a specified dimension of the input,
or of the entire input. The Variance block can also track the variance
of a sequence of inputs over a period of time. The **Running variance**
parameter selects between basic operation and running operation.

Basic Operation

When you do not select the **Running variance** check box, the block
computes the variance of each row or column of the input, along vectors
of a specified dimension of the input, or of the entire input at each
individual sample time, and outputs the array y. Each element in y is
the variance of the corresponding column, row, vector, or entire input.
The output y depends on the setting of the **Find the variance value
over** parameter. For example, consider a 3-dimensional input signal
of size M-by-N-by-P:

- Entire input — The output at each sample time is a scalar that
  contains the variance of the entire input.

  ```
  y = var(u(:))     % Equivalent MATLAB code
  ```

- Each row — The output at each sample time consists of an
  M-by-1-by-P array, where each element contains the variance of each
  vector over the second dimension of the input. For an input that
  is an M-by-N matrix, the output at each sample time is an M-by-1
  column vector.

  ```
  y = var(u,0,2)    % Equivalent MATLAB code
  ```

- Each column — The output at each sample time consists of a
  1-by-N-by-P array, where each element contains the variance of each

vector over the first dimension of the input. For an input that is an *M*-by-*N* matrix, the output at each sample time is a 1-by-*N* row vector.

```
y = var(u,0,1)      % Equivalent MATLAB code
```

In this mode, the block treats length-*M* unoriented vector inputs as *M*-by-1 column vectors.

- `Specified dimension` — The output at each sample time depends on **Dimension**. If **Dimension** is set to 1, the output is the same as that when you select `Each column`. If **Dimension** is set to 2, the output is the same as when you select `Each row`. If **Dimension** is set to 3, the output at each sample time is an *M*-by-*N* matrix containing the variance of each vector over the third dimension of the input.

```
y = var(u,0,Dimension)      % Equivalent MATLAB code
```

For purely real or purely imaginary inputs, the variance of an M-by-N matrix is the square of the standard deviation:

$$y = \sigma^2 = \frac{\sum\limits_{i=1}^{M}\sum\limits_{j=1}^{N}\left|u_{ij}\right|^2 - \dfrac{\left|\sum\limits_{i=1}^{M}\sum\limits_{j=1}^{N}u_{ij}\right|^2}{M*N}}{M*N-1}$$

For complex inputs, the variance is given by the following equation:

$$\sigma^2 = {\sigma_{Re}}^2 + {\sigma_{Im}}^2$$

Running Operation

When you select the **Running variance** check box, the block tracks the variance of successive inputs to the block. In this mode, you must also specify a value for the **Input processing** parameter:

- When you select Elements as channels (sample based), the block outputs an *M*-by-*N* array. Each element y_{ij} of the output contains the variance of the element u_{ij} over all inputs since the last reset.

- When you select Columns as channels (frame based), the block outputs an *M*-by-*N* matrix. Each element y_{ij} of the output contains the variance of the *j*th column over all inputs since the last reset, up to and including element u_{ij} of the current input.

Running Operation for Variable-Size Inputs

When your inputs are of variable size, and you select the **Running variance** check box, there are two options:

- If you set the **Input processing** parameter to Elements as channels (sample based), the state is reset.

- If you set the **Input processing** parameter to Columns as channels (frame based), then there are two cases:

  - When the input size difference is in the number of channels (i.e., number of columns), the state is reset.

  - When the input size difference is in the length of channels (i.e., number of rows), there is no reset and the running operation is carried out as usual.

Variance

Resetting the Running Variance

The block resets the running variance whenever a reset event is detected at the optional Rst port. The reset sample time must be a positive integer multiple of the input sample time.

You specify the reset event in the **Reset port** parameter:

- None disables the Rst port.

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

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

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

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

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

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

Falling edge   Falling edge

Falling edge   Falling edge   **Not a falling edge** because it continues
                                            a fall from a positive value to zero.

- `Either edge` — Triggers a reset operation when the Rst input is a `Rising edge` or `Falling edge` (as described earlier)

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

---

Note When running simulations in the Simulink MultiTasking mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and the topic on models with multiple sample rates in the Simulink Coder documentation.

ROI Processing

To calculate the statistical value within a particular region of interest (ROI) of the input, select the **Enable ROI processing** check box. This option is only available when the **Find the variance value over** parameter is set to `Entire input` and the **Running variance** check box is not selected. ROI processing is only supported for 2-D inputs.

Variance

Use the **ROI type** parameter to specify whether the ROI is a binary mask, label matrix, rectangle, or line. ROI processing is only supported for 2-D inputs.

- A binary mask is a binary image that enables you to specify which pixels to highlight, or select.

- In a label matrix, pixels equal to 0 represent the background, pixels equal to 1 represent the first object, pixels equal to 2 represent the second object, and so on. When the **ROI type** parameter is set to Label matrix, the Label and Label Numbers ports appear on the block. Use the Label Numbers port to specify the objects in the label matrix for which the block calculates statistics. The input to this port must be a vector of scalar values that correspond to the labeled regions in the label matrix.

- For more information about the format of the input to the ROI port when the ROI is a rectangle or a line, see the Draw Shapes reference page.

Note For rectangular ROIs, use the **ROI portion to process** parameter to specify whether to calculate the statistical value for the entire ROI or just the ROI perimeter.

Use the **Output** parameter to specify the block output. The block can output separate statistical values for each ROI or the statistical value for all specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select Binary mask.

If, for the **ROI type** parameter, you select `Rectangles` or `Lines`, the **Output flag indicating if ROI is within image bounds** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

Output = Individual Statistics for Each ROI

Flag Port Output	Description
0	ROI is completely outside the input image.
1	ROI is completely or partially inside the input image.

Output = Single Statistic for All ROIs

Flag Port Output	Description
0	All ROIs are completely outside the input image.
1	At least one ROI is completely or partially inside the input image.

If the ROI is partially outside the image, the block only computes the statistical values for the portion of the ROI that is within the image.

If, for the **ROI type** parameter, you select `Label matrix`, the **Output flag indicating if input label numbers are valid** check box appears in the dialog box. If you select this check box, the Flag port appears on the block. The following tables describe the Flag port output based on the block parameters.

Variance

Output = Individual Statistics for Each ROI

Flag Port Output	Description
0	Label number is not in the label matrix.
1	Label number is in the label matrix.

Output = Single Statistic for All ROIs

Flag Port Output	Description
0	None of the label numbers are in the label matrix.
1	At least one of the label numbers is in the label matrix.

Fixed-Point Data Types

The parameters on the **Data Types** pane of the block dialog are only used for fixed-point inputs. For purely real or purely imaginary inputs, the variance of the input is the square of its standard deviation. For complex inputs, the output is the sum of the variance of the real and imaginary parts of the input.

The following diagram shows the data types used within the Variance block for fixed-point signals.
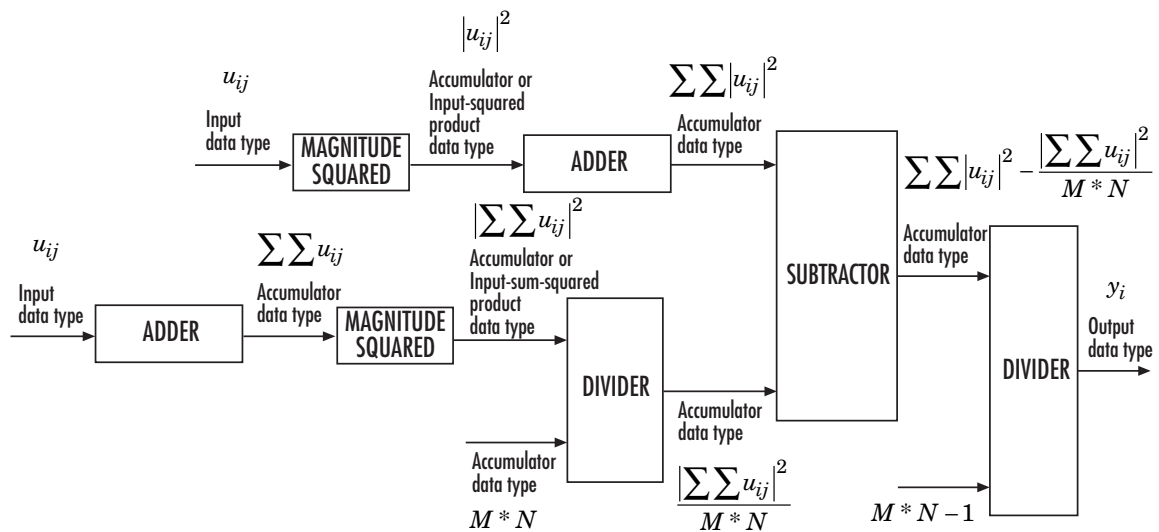
The results of the magnitude-squared calculations in the figure are in the product output data type. You can set the accumulator, product output, and output data types in the block dialog as discussed in "Dialog Box" on page 2-1693.

Examples   In the following ex_variance_ref model, the Variance block calculates the running variance of a 3-by-2 matrix input, u. The **Input processing** parameter is set to Columns as channels (frame based), so the block processes the input as a two channel signal with a frame size of three. The running variance is reset at t=2 by an impulse to the block's Rst port.

Variance
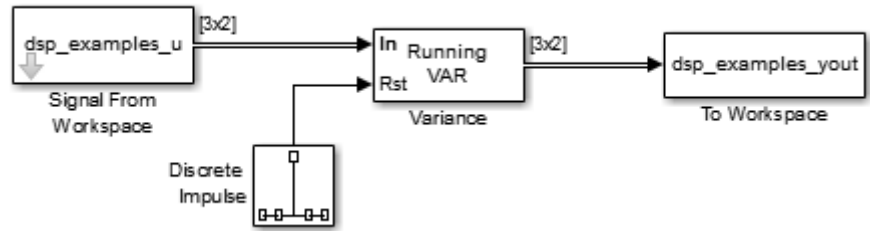
The operation of the block is shown in the following figure.
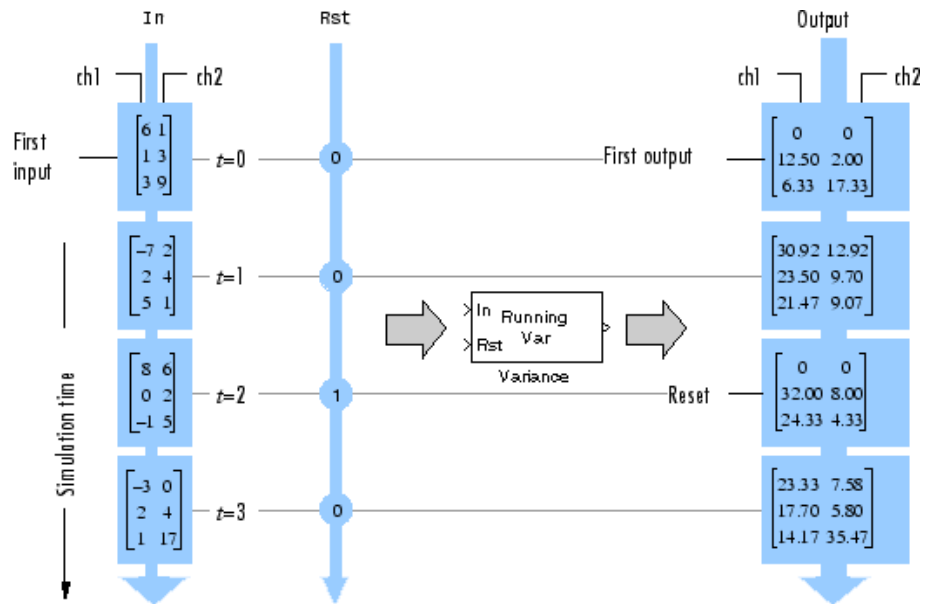
Dialog Box

The **Main** pane of the Variance block dialog appears as follows.
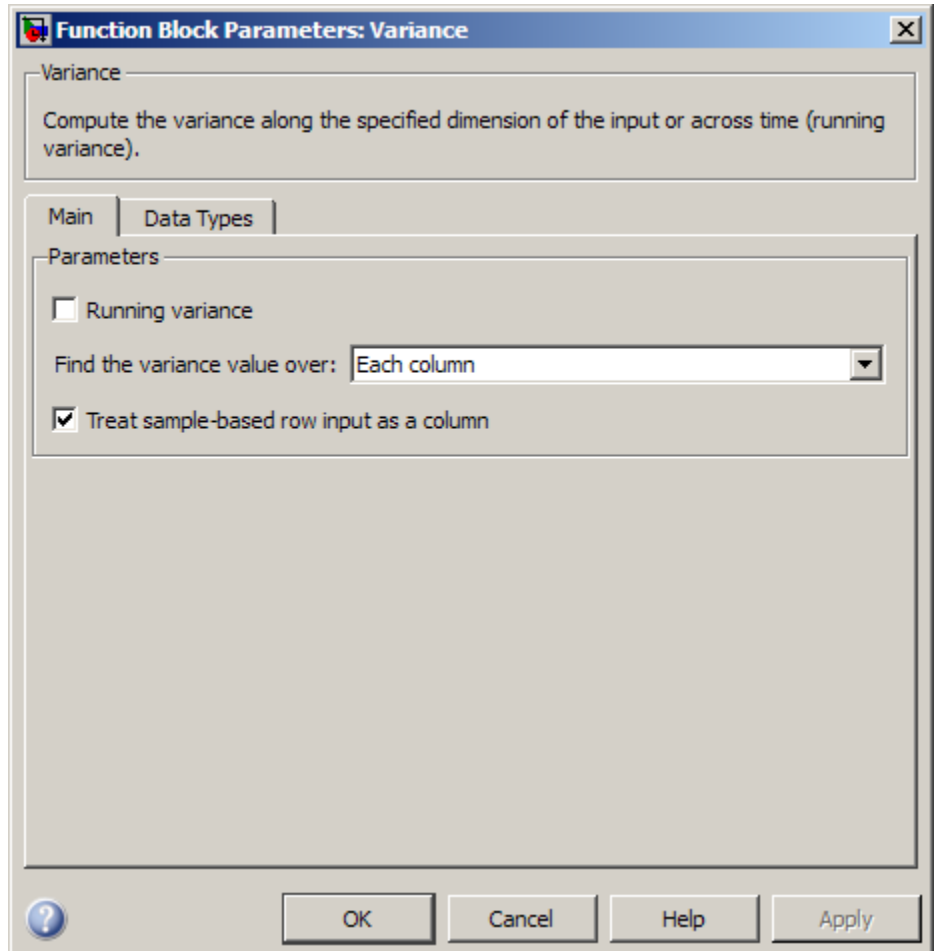
Running variance
  Enables running operation when selected.

Variance

Input processing

Specify how the block should process the input when computing the running variance. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

This parameter appears only when you select the **Running variance** check box.

Note The option Inherit from input (this choice will be removed - see release notes) will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Reset port

Specify the reset event that causes the block to reset the running variance. The sample time of the input to the Rst port must be a positive integer multiple of the input sample time. This parameter appears only when you select the **Running variance** check box. For more information, see "Resetting the Running Variance" on page 2-1686

Find the variance value over

Specify whether to find the variance along rows, columns, entire input, or the dimension specified in the **Dimension** parameter. For more information, see "Basic Operation" on page 2-1683.

Treat sample-based row input as a column

Select to treat sample-based length-*M* row vector inputs as *M*-by-1 column vectors. This parameter is only visible when the **Find the variance value over** parameter is set to Each column.

Note This check box will be removed in a future release. See "Sample-Based Row Vector Processing Changes" in the DSP System Toolbox Release Notes.

Dimension

Specify the dimension (one-based value) of the input signal, over which the variance is computed. The value of this parameter cannot exceed the number of dimensions in the input signal. This parameter is only visible when the **Find the variance value over** parameter is set to Specified dimension.

Enable ROI Processing

Select this check box to calculate the statistical value within a particular region of each image. This parameter is only available when the **Find the variance value over** parameter is set to Entire input, and the block is not in running mode.

Note Full ROI processing is available only if you have a Computer Vision System Toolbox license. If you do not have a Computer Vision System Toolbox license, you can still use ROI processing, but are limited to the **ROI type** Rectangles.

ROI type

Specify the type of ROI you want to use. Your choices are Rectangles, Lines, Label matrix, or Binary mask.

Variance

ROI portion to process

Specify whether you want to calculate the statistical value for the entire ROI or just the ROI perimeter. This parameter is only visible if, for the **ROI type** parameter, you specify Rectangles.

Output

Specify the block output. The block can output a vector of separate statistical values for each ROI or a scalar value that represents the statistical value for all the specified ROIs. This parameter is not available if, for the **ROI type** parameter, you select Binary mask.

Output flag

☑ **Output flag indicating if ROI is within image bounds**

☑ **Output flag indicating if label numbers are valid**

When you select either of these check boxes, the Flag port appears on the block. For a description of the Flag port output, see the tables in "ROI Processing" on page 2-1687.

The **Output flag indicating if ROI is within image bounds** check box is only visible when you select Rectangles or Lines as the **ROI type**.

The **Output flag indicating if label numbers are valid** check box is only visible when you select Label matrix for the **ROI type** parameter.

The **Data Types** pane of the Variance block dialog appears as follows.
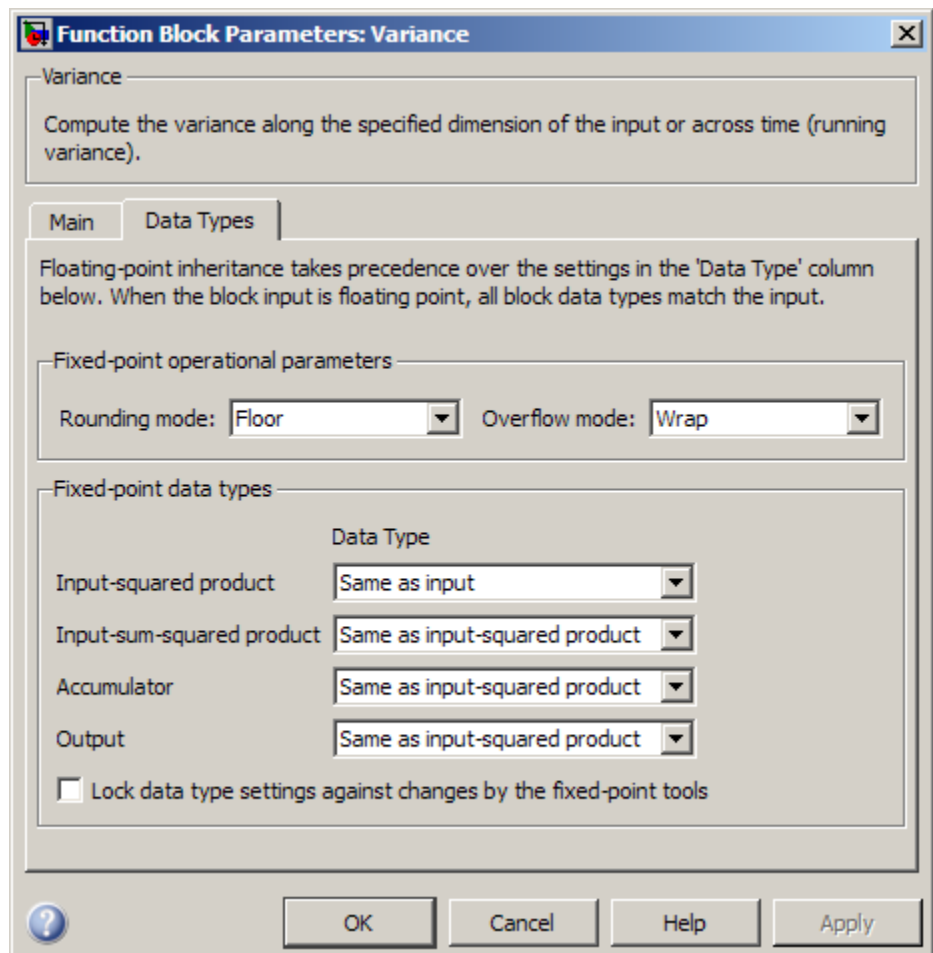
Rounding mode
   Select the rounding mode for fixed-point operations.

Overflow mode
   Select the overflow mode for fixed-point operations.

Variance

Input-squared product

Use this parameter to specify how to designate the input-squared
product word and fraction lengths:

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

- When you select `Binary point scaling`, you can enter the
  word length and the fraction length of the input-squared
  product, in bits.

- When you select `Slope and bias scaling`, you can enter the
  word length, in bits, and the slope of the input-squared product.
  This block requires power-of-two slope and a bias of zero.

Input-sum-squared product

Use this parameter to specify how to designate the
input-sum-squared product word and fraction lengths:

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

- When you select `Binary point scaling`, you can enter the
  word length and the fraction length of the input-sum-squared
  product, in bits.

- When you select `Slope and bias scaling`, you can enter the
  word length, in bits, and the slope of the input-sum-squared
  product. This block requires power-of-two slope and a bias of
  zero.

Accumulator

Use this parameter to specify the accumulator word and fraction
lengths resulting from a complex-complex multiplication in the
block:

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

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

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

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

Output

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

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

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

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

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

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

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Variance

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Reset	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
ROI	Rectangles and lines: • Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Binary Mask: • Boolean
Label	• 8-, 16-, and 32-bit unsigned integers
Label Numbers	• 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Flag	• Boolean

See Also

Mean	DSP System Toolbox
RMS	DSP System Toolbox
Standard Deviation	DSP System Toolbox
var	MATLAB

Vector Quantizer Decoder

Purpose   Find vector quantizer codeword that corresponds to given, zero-based index value

Library   Quantizers

dspquant2

Description

Vector Quantizer
Decoder
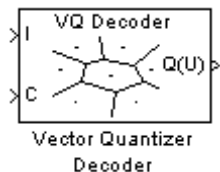
The Vector Quantizer Decoder block associates each input index value with a codeword, a column vector of quantized output values defined in the **Codebook values** parameter. When you input multiple index values into this block, the block outputs a matrix of quantized output vectors. This matrix is created by horizontally concatenating the codeword vectors that correspond to each index value.

You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select Specify via dialog, you can type the codebook values into the block parameters dialog box. Select Input port and port C appears on the block. The block uses the input to port C as the **Codebook values** parameter.

The **Codebook values** parameter is a k-by-N matrix of values, where $k \geq 1$ and $N \geq 1$. Each column of this matrix is a codeword vector, and each codeword vector corresponds to an index value. The index values are zero based; therefore, the first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on.

The input to this block is a vector of index values, where $0 \leq index < N$ and N is the number of columns of the codebook matrix. Use the **Action for out of range index value** parameter to determine how the block behaves when an input index value is out of this range. When you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N-1, select Clip. When you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to N are set to N-1, select Clip and warn. When you want the simulation to stop and display an error when the index values are out of range, select Error.

Data Type Support

The input to the block can be the index values and the codebook values. The data type of the index input to the block at port I can be uint8, uint16, uint32, int8, int16, or int32. The data type of the codebook values can be double, single, or Fixed-point.

The output of the block is the quantized output values. These quantized output values always have the same data type as the codebook values. When the codebook values are specified via an input port, the block assigns the same data type to the Q(U) output port. When the codebook values are specified via the dialog, use the **Codebook and output data type** parameter to specify the data type of the Q(U) output port. The data type of the codebook and quantized output can be Same as input, double, single, Fixed-point, User-defined, or Inherit via back propagation.

Vector Quantizer Decoder

<table>
<tr><td>**Dialog Box**</td><td>The **Main** pane of the Vector Quantizer Decoder block dialog appears as follows.</td></tr>
</table>
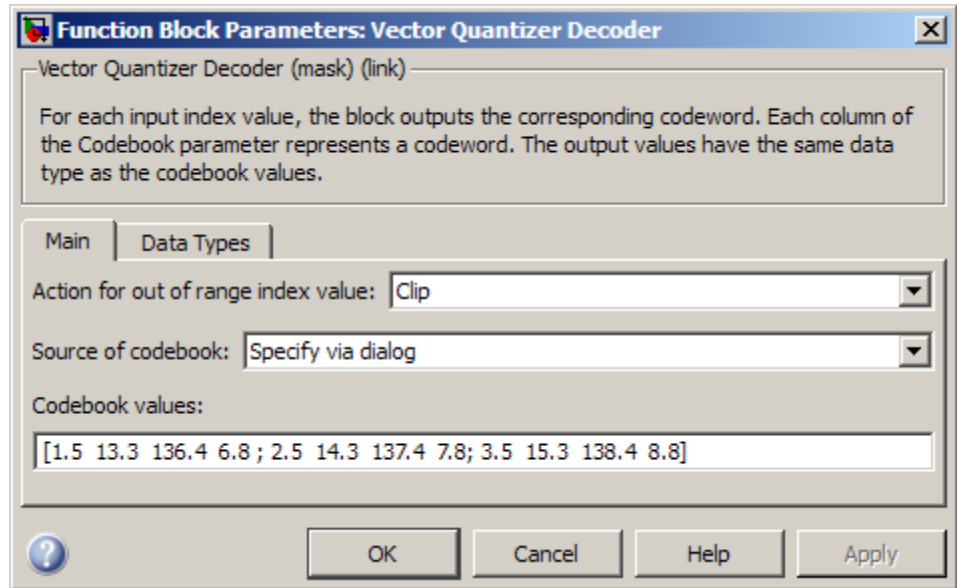
Action for out of range index value

Choose the behavior of the block when an input index value is out of range, where $0 \leq index < N$ and N is the length of the codebook vector. Select `Clip` when you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N-1. Select `Clip and warn` when you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to N are set to N-1. Select `Error` when you want the simulation to stop and display an error when the index values are out of range.

Source of codebook

Choose `Specify via dialog` to type the codebook values into the block parameters dialog box. Select `Input port` to specify the codebook values using the block's input port, C.

Codebook values

Enter a k-by-N matrix of quantized output values, where $1 \leq k$ and $1 \leq N$. Each column of your matrix corresponds to an index value. This parameter is visible if, from the **Source of codebook** list, you select Specify via dialog.

The **Data Types** pane of the Vector Quantizer Decoder block dialog appears as follows.
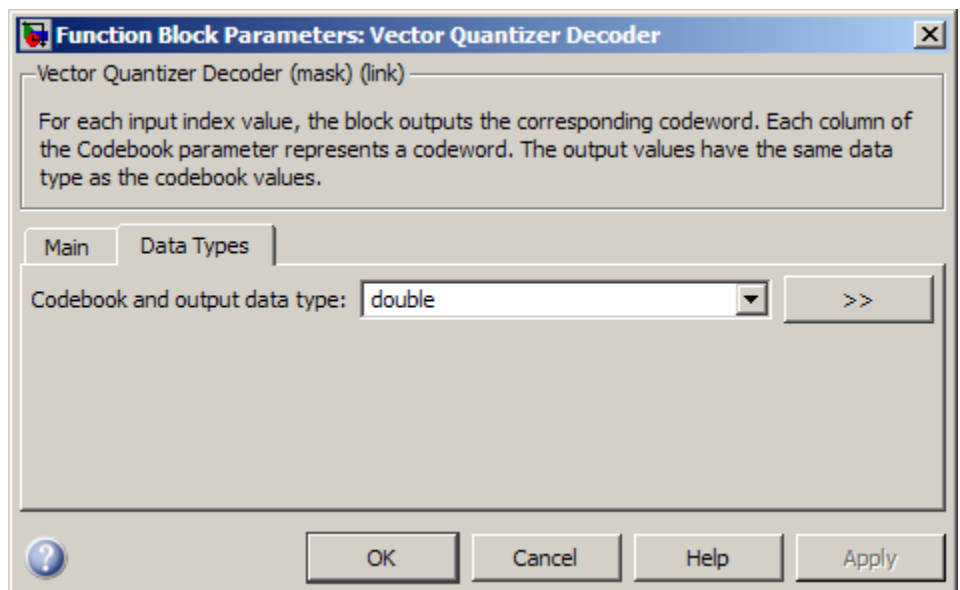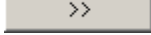
Codebook and output data type

Specify the data type of the codebook and quantized output values. You can select one of the following:

- A rule that inherits a data type, for example, Inherit: Same as input.

- A built in data type, such as double

Vector Quantizer Decoder

- An expression that evaluates to a valid data type, for example,
  `fixdt(1,16)`

Click the **Show data type assistant** button [>>] to
display the **Data Type Assistant**, which helps you set the
Output data type parameter.

See "Specify Block Output Data Types" in *Simulink User's Guide*
for more information.

This parameter is available only when you set the **Source of
codebook** parameter to `Specify via dialog`. If you set the
Source of codebook parameter to `Input port`, the output
values have the same data type as the input codebook values.

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression.*
Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

Port	Supported Data Types
I	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
C	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
Q(U)	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Design	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox
Vector Quantizer Encoder	DSP System Toolbox

Vector Quantizer Design

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

Library   Quantizers

       dspquant2

Description

Vector Quantizer
Design
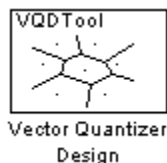
Double-click on the Vector Quantizer Design block to start VQDTool, a GUI that allows you to design and implement a vector quantizer. You can also start VQDTool by typing vqdtool at the MATLAB command prompt. Based on your specifications, VQDTool iteratively calculates the codebook values that minimize the mean squared error between the training set and the codebook until the stopping criteria for the design process is satisfied. The block uses the resulting codebook values to implement your vector quantizer.

For the **Training Set** parameter, enter a k-by-M matrix of values you want to use to train the quantizer codebook. The variable k, where $k \geq 1$, is the length of each training vector. It also represents the dimension of your quantizer. The variable M, where $M \geq 2$, is the number of training vectors. This data can be created using a MATLAB function, such as the default value randn(10,1000), or it can be any variable defined in the MATLAB workspace.

You have two choices for the **Source of initial codebook** parameter. Select Auto-generate to have the block choose the values of the initial codebook. In this case, the block picks N random training vectors as the initial codebook, where N is the **Number of levels** parameter and $N \geq 2$. When you select User defined, enter the initial codebook values in the **Initial codebook** field. The initial codebook matrix must have the same number of rows as the training set. Each column of the codebook is a codeword, and your codebook must have at least two codewords.

For the given training set and initial codebook, the block performs an iterative process, using the Generalized Lloyd Algorithm (GLA), to design a final codebook. For each iteration of the GLA, the block first associates each training vector with its nearest codeword by calculating

the distortion. You can specify one of the two possible methods for calculating distortion using the **Distortion measure** parameter.

When you select `Squared error` for the **Distortion measure** parameter, the block finds the nearest codeword by calculating the squared error (unweighted). Consider the codebook

$CB = \begin{bmatrix} CW_1 & CW_2 & ... & CW_N \end{bmatrix}$. This codebook has N codewords; each codeword has k elements. The i-th codeword is defined as

$CW_i = \begin{bmatrix} a_{1i} & a_{2i} & ... & a_{ki} \end{bmatrix}$. The training set has M columns and is

defined as $U = \begin{bmatrix} U_1 & U_2 & ... & U_M \end{bmatrix}$, where the p-th training vector

is $U_p = \begin{bmatrix} u_{1p} & u_{2p} & ... & u_{kp} \end{bmatrix}'$. The squared error (unweighted) is calculated using the equation

$$D = \sum_{j=1}^{k} \left(a_{ji} - u_{jp} \right)^2$$

When you select `Weighted squared error` for the **Distortion measure** parameter, enter a vector or matrix for the **Weighting factor** parameter. When the weighting factor is a vector, its length must be equal to the number of rows in the training set. This weighting factor is used for each training vector. When the weighting factor is a matrix, it must be the same size as the training set matrix. The block finds the nearest codeword by calculating the weighted squared error. If

the weighting factor for the p-th column of the training vector, U_p,

is defined as $Wp = \begin{bmatrix} w_{1p} & w_{2p} & ... & w_{kp} \end{bmatrix}'$, then the weighted squared error is defined by the equation

$$D = \sum_{j=1}^{k} w_{jp} \left(a_{ji} - u_{jp} \right)^2$$

Once the block has associated all the training vectors with their nearest codeword vectors, the block calculates the mean squared error for the

codebook and checks to see if the stopping criteria for the process has been satisfied.

The two possible options for the **Stopping criteria** parameter are `Relative threshold` and `Maximum iteration`. When you want the design process to stop when the fractional drop in the squared error is below a certain value, select `Relative threshold`. Then, type the maximum acceptable fractional drop in the **Relative threshold** field. The fraction drop in the squared error is defined as

$$\frac{\text{error at previous iteration} - \text{error at current iteration}}{\text{error at previous iteration}}$$

When you want the design process to stop after a certain number of iterations, choose `Maximum iteration`. Then, enter the maximum number of iterations you want the block to perform in the **Maximum iteration** field. For **Stopping criteria**, you can also choose `Whichever comes first` and enter **Relative threshold** and **Maximum iteration** values. The block stops iterating as soon as one of these conditions is satisfied.

When a training vector has the same distortion for two different codeword vectors, the algorithm uses the **Tie-breaking rule** parameter to determine which codeword vector the training vector is associated with. When you want the training vector to be associated with the lower indexed codeword, select `Lower indexed codeword`. To associate the training vector with the higher indexed codeword, select `Higher indexed codeword`.

With each iteration, the block updates the codeword values in order to minimize the distortion. The **Codebook update method** parameter defines the way the block calculates these new codebook values.

Note If, for the **Distortion measure** parameter, you choose `Squared error`, the **Codebook update method** parameter is set to `Mean`.

If, for the **Distortion measure** parameter, you choose `Weighted squared error` and you choose `Mean` for the **Codebook update method** parameter, the new codeword vector is found as follows. Suppose there are three training vectors associated with one codeword vector. The training vectors are

$$TS_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \ TS_3 = \begin{bmatrix} 10 \\ 12 \end{bmatrix}, \text{ and } TS_7 = \begin{bmatrix} 11 \\ 12 \end{bmatrix}.$$

The new codeword vector is calculated as

$$CW_{new} = \begin{bmatrix} \dfrac{1+10+11}{3} \\ \dfrac{2+12+12}{3} \end{bmatrix}$$

where the denominator is the number of training vectors associated with this codeword. If, for the **Codebook update method** parameter,

you choose `Centroid` and you specify the weighting factors $W_1 = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$,

$W_3 = \begin{bmatrix} 1 \\ 0.6 \end{bmatrix}$, and $W_7 = \begin{bmatrix} 0.3 \\ 0.4 \end{bmatrix}$, the new codeword vector is calculated as

$$CW_{new} = \begin{bmatrix} \dfrac{(0.1)(1)+(1)(10)+(0.3)(11)}{0.1+1+0.3} \\ \dfrac{(0.2)(2)+(0.6)(12)+(0.4)(12)}{0.2+0.6+0.4} \end{bmatrix}$$

Click **Design and Plot** to design the quantizer with the parameter values specified on the left side of the GUI. The performance curve and the entropy of the quantizer are updated and displayed in the figures on the right side of the GUI.

Vector Quantizer Design

> **Note** You must click **Design and Plot** to apply any changes you make to the parameter values in the VQDTool dialog box.

The following is an example of how the block calculates the entropy of the quantizer at each iteration. Suppose you have a codebook with four codewords and a training set with 200 training vectors. Also suppose that, at the i-th iteration, 40 training vectors are associated with the first codeword, 60 training vectors are associated with the second codeword, 20 training vectors are associated with the third codeword, and 80 training vectors are associated with the fourth codeword. The probability that a training vector is associated with the first codeword is $\dfrac{40}{200}$. The probabilities that training vectors are associated with the second, third, and fourth codewords are $\dfrac{60}{200}$, $\dfrac{20}{200}$, and $\dfrac{80}{200}$, respectively. The GUI uses these probabilities to calculate the entropy according to the equation

$$H = \sum_{i=1}^{N} -p_i \log_2 p_i$$

where N is the number of codewords. Based on these probabilities, the GUI calculates the entropy of the quantizer at the i-th iteration as

$$H = -\left(\frac{40}{200} \log_2 \frac{40}{200} + \frac{60}{200} \log_2 \frac{60}{200} + \frac{20}{200} \log_2 \frac{20}{200} + \frac{80}{200} \log_2 \frac{80}{200} \right)$$
$$H = 1.8464$$

VQDTool can export parameter values that correspond to the figures displayed in the GUI. Click the **Export Outputs** button, or press **Ctrl+E**, to export the **Final Codebook**, **Mean Square Error**, and **Entropy** values to the workspace, a text file, or a MAT-file.

In the **Model** section of the GUI, specify the destination of the block that will contain the parameters of your quantizer. For **Destination**, select `Current model` to create a block with your parameters in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Select `New model` to create a block in a new model file.

From the **Block type** list, select `Encoder` to design a Vector Quantizer Encoder block. Select `Decoder` to design a Vector Quantizer Decoder block. Select `Both` to design a Vector Quantizer Encoder block and a Vector Quantizer Decoder block.

In the **Encoder block name** field, enter a name for the Vector Quantizer Encoder block. In the **Decoder block name** field, enter a name for the Vector Quantizer Decoder block. When you have a Vector Quantizer Encoder and/or Decoder block in your destination model with the same name, select the **Overwrite target block** check box to replace the block's parameters with the current parameters. When you do not select this check box, a new Vector Quantizer Encoder and/or Decoder block is created in your destination model.

Click **Generate Model**. VQDTool uses the parameters that correspond to the current plots to set the parameters of the Vector Quantizer Encoder and/or Decoder blocks.

Vector Quantizer Design

Dialog
Box

Training Set

Enter the samples of the signal you would like to quantize. This data set can be a MATLAB function or a variable defined in the MATLAB workspace. The typical length of this data vector is `1e5`.

Source of initial codebook

Select `Auto-generate` to have the block choose the initial codebook values. Choose `User defined` to enter your own initial codebook values.

Number of levels

Enter the number of codeword vectors, N, in your codebook matrix, where $N \geq 2$.

Initial codebook

Enter your initial codebook values. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter. The codebook must have the same number of rows as the training set. You must provide at least two codeword vectors.

Distortion measure

When you select `Squared error`, the block finds the nearest codeword by calculating the squared error (unweighted). When you select `Weighted squared error`, the block finds the nearest codeword by calculating the weighted squared error.

Weighting factor

Enter a vector or matrix. The block uses these values to compute the weighted squared error. When the weighting factor is a vector, its length must be equal to the number of rows in the training set. This weighting factor is used for each training vector. When the weighting factor is a matrix, it must be the same size as the training set matrix. The individual weighting factors cannot be negative. The weighting factor vector or matrix cannot contain all zeros.

Stopping criteria

Choose `Relative threshold` to enter the maximum acceptable fractional drop in the squared quantization error. Choose `Maximum iteration` to specify the number of iterations at which to stop.

Choose `Whichever comes first` and the block stops the iteration process as soon as the relative threshold or maximum iteration value is attained.

Relative threshold

This parameter is available when you choose `Relative threshold` or `Whichever comes first` for the **Stopping criteria** parameter. Enter the value that is the maximum acceptable fractional drop in the squared quantization error.

Maximum iteration

This parameter is available when you choose `Maximum iteration` or `Whichever comes first` for the **Stopping criteria** parameter. Enter the maximum number of iterations you want the block to perform.

Tie-breaking rules

When a training vector has the same distortion for two different codeword vectors, select `Lower indexed codeword` to associate the training vector with the lower indexed codeword. Select `Higher indexed codeword` to associate the training vector with the lower indexed codeword.

Codebook update method

When you choose `Mean`, the new codeword vector is calculated by taking the average of all the training vector values that were associated with the original codeword vector. When you choose `Centroid`, the block calculates the new codeword vector by taking the weighted average of all the training vector values that were associated with the original codeword vector Note that if, for the **Distortion measure** parameter, you choose `Squared error`, the **Codebook update method** parameter is set to `Mean`.

Destination

Choose `Current model` to create a Vector Quantizer block in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Choose `New model` to create a block in a new model file.

Block type

Select `Encoder` to design a Vector Quantizer Encoder block. Select `Decoder` to design a Vector Quantizer Decoder block. Select `Both` to design a Vector Quantizer Encoder block and a Vector Quantizer Decoder block.

Encoder block name

Enter a name for the Vector Quantizer Encoder block.

Decoder block name

Enter a name for the Vector Quantizer Decoder block.

Overwrite target block

When you do not select this check box and a Vector Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, a new Vector Quantizer Encoder and/or Decoder block is created in the destination model. When you select this check box and a Vector Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, the parameters of these blocks are overwritten by new parameters.

Generate Model

Click this button and VQDTool uses the parameters that correspond to the current plots to set the parameters of the Vector Quantizer Encoder and/or Decoder blocks.

Design and Plot

Click this button to design a quantizer using the parameters on the left side of the GUI and to update the performance curve and entropy plots on the right side of the GUI.

You must click **Design and Plot** to apply any changes you make to the parameter values in the VQDTool GUI.

Export Outputs

Click this button, or press **Ctrl+E**, to export the **Final Codebook**, **Mean Squared Error**, and **Entropy** values to the workspace, a text file, or a MAT-file.

Vector Quantizer Design

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression.* Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

- Double-precision floating point

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Design	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox
Vector Quantizer Decoder	DSP System Toolbox
Vector Quantizer Encoder	DSP System Toolbox

Purpose   For given input, find index of nearest codeword based on Euclidean or weighted Euclidean distance measure

Library   Quantizers

dspquant2
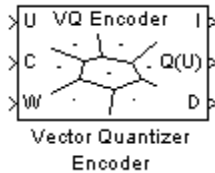
Description   The Vector Quantizer Encoder block compares each input column vector to the codeword vectors in the codebook matrix. Each column of this codebook matrix is a codeword. The block finds the codeword vector nearest to the input column vector and returns its zero-based index. This block supports real floating-point and fixed-point signals on all input ports.

The block finds the nearest codeword by calculating the distortion. The block uses two methods for calculating distortion: Euclidean squared error (unweighted) and weighted Euclidean squared error. Consider the codebook, $CB = \begin{bmatrix} CW_1 & CW_2 & ... & CW_N \end{bmatrix}$. This codebook has N codewords; each codeword has k elements. The i-th codeword is defined as a column vector, $CW_i = \begin{bmatrix} a_{1i} & a_{2i} & ... & a_{ki} \end{bmatrix}$. The multichannel input has M columns and is defined as $U = \begin{bmatrix} U_1 & U_2 & ... & U_M \end{bmatrix}$, where the p-th input column vector is $U_p = \begin{bmatrix} u_{1p} & u_{2p} & ... & u_{kp} \end{bmatrix}'$. The squared error (unweighted) is calculated using the equation

$$D = \sum_{j=1}^{k} \left(a_{ji} - u_{jp} \right)^2$$

The weighted squared error is calculated using the equation

$$D = \sum_{j=1}^{k} w_j \left(a_{ji} - u_{jp} \right)^2$$

where the weighting factor is defined as $W = \begin{bmatrix} w_1 & w_2 & ... & w_k \end{bmatrix}$. The index of the codeword that is associated with the minimum distortion is assigned to the input column vector.

You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select `Specify via dialog`, you can type the codebook values into the block parameters dialog box. Select `Input port` and port C appears on the block. The block uses the input to port C as the **Codebook** parameter.

The **Codebook** parameter is an k-by-N matrix of values, where $k \geq 1$ and $N \geq 1$. Each input column vector is compared to this codebook. Each column of the codebook matrix is a codeword, and each codeword has an index value. The first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on. The codeword vectors must have the same number of rows as the input, U.

For the **Distortion measure** parameter, select `Squared error` when you want the block to calculate the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. Select `Weighted squared error` when you want to use a weighting factor to emphasize or deemphasize certain input values.

For the **Source of weighting factor** parameter, select `Specify via dialog` to enter a weighting factor vector in the dialog box. Choose `Input port` to specify the weighting factor using port W.

Use the **Weighting factor** parameter to emphasize or deemphasize certain input values when calculating the distortion measure. For

example, consider the p-th input column vector, U_p, as previously defined. When you want to neglect the effect of the first element of this vector, enter [0 1 1 ... 1] as the **Weighting factor** parameter. This weighting factor is used to calculate the weighted squared error using the equation

$$D = \sum_{j=1}^{k} w_j \left(a_{ji} - u_{jp} \right)^2$$

Because of the weighting factor used in this example, the weighted squared error is not affected by the first element of the input matrix. Therefore, the first element of the input column vector no longer impacts the choice of index value output by the Vector Quantizer Encoder block.

Use the **Index output data type** parameter to specify the data type of the index values output at port I. The data type of the index values can be `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

When an input vector is equidistant from two codewords, the block uses the **Tie-breaking rule** parameter to determine which index value the block chooses. When you want the input vector to be represented by the lower index valued codeword, select `Choose the lower index`. To represent the input column vector by the higher index valued codeword, select `Choose the higher index`.

Select the **Output codeword** check box to output at port Q(U) the codeword vectors that correspond to each index value. When the input is a matrix, the corresponding codeword vectors are horizontally concatenated into a matrix.

Select the **Output quantization error** check box to output at port D the quantization error that results when the block represents the input column vector by its nearest codeword. When the input is a matrix, the quantization error values are horizontally concatenated.

The Vector Quantizer Encoder block accepts real floating-point and fixed-point inputs. For more information on the data types accepted by each port, see "Data Type Support" on page 2-1721 or "Supported Data Types" on page 2-1728.

Data Type Support

The input data values, codebook values, and weighting factor values are input to the block at ports U, C, and W, respectively. The data type of the input data values, codebook values, and weighting factor values can

be double, single, or Fixed-point. The input data, codebook values, and weighting factor must be the same data type.

The outputs of the block are the index values, output codewords, and quantization error. Use the **Index output data type** parameter to specify the data type of the index output from the block at port I. The data type of the index can be int8, uint8, int16, uint16, int32, or uint32. The data type of the output codewords and the quantization error can be double, single, or Fixed-point. The block assigns the data type of the output codewords and the quantization error based on the data type of the input data.

Fixed-Point Data Types

The following diagram shows the data types used within the Vector Quantizer Encoder block for fixed-point signals.
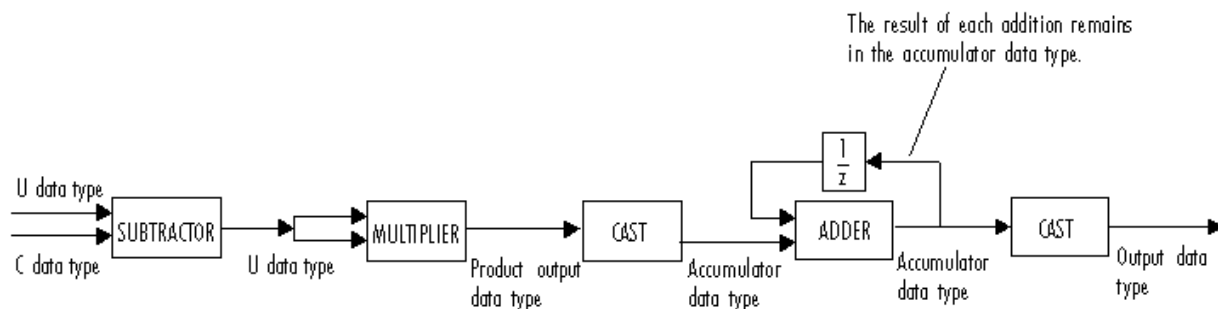
You can set the product output, accumulator, and index output data types in the block dialog as discussed below.

Dialog Box

The **Main** pane of the Vector Quantizer Encoder block dialog appears as follows.
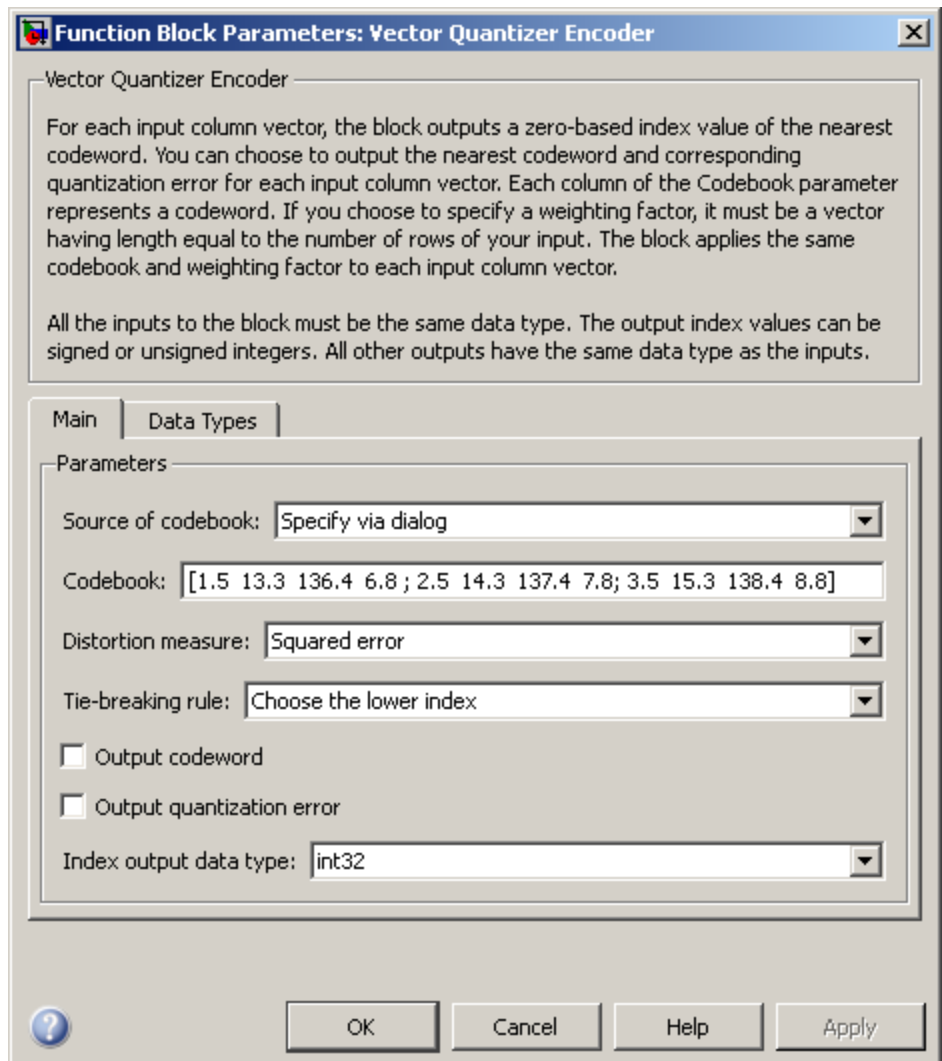
Vector Quantizer Encoder

Source of codebook

Choose `Specify via dialog` to type the codebook values into the block parameters dialog box. Select `Input port` to specify the codebook values using the block's input port, C.

Codebook

Enter a k-by-N matrix of values, where $1 \leq k$ and $1 \leq N$, to which your input column vector or matrix is compared. This parameter is visible if, from the **Source of codebook** list, you select `Specify via dialog`.

Distortion measure

Select `Squared error` when you want the block to calculate the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. Select `Weighted squared error` when you want the block to calculate the distortion by evaluating a weighted Euclidean distance using a weighting factor to emphasize or deemphasize certain input values.

Source of weighting factor

Select `Specify via dialog` to enter a value for the weighting factor in the dialog box. Choose `Input port` and specify the weighting factor using port W on the block. This parameter is visible if, for the **Distortion measure** parameter, you select `Weighted squared error`.

Weighting factor

Enter a vector of values. This vector must have length equal to the number of rows of the input, U. This parameter is visible if, for the **Source of weighting factor** parameter, you select `Specify via dialog`.

Tie-breaking rule

Set this parameter to determine the behavior of the block when an input column vector is equidistant from two codewords. When you want the input column vector to be represented by the lower index valued codeword, select `Choose the lower index`. To represent

the input column vector by the higher index valued codeword, select Choose the higher index.

Output codeword

Select this check box to output the codeword vectors nearest to the input column vectors.

Output quantization error

Select this check box to output the quantization error value that results when the block represents the input column vector by the nearest codeword.

Index output data type

Select int8, uint8, int16, uint16, int32, or uint32 as the data type of the index output at port I.

The **Data Types** pane of the Vector Quantizer Encoder block dialog appears as follows.

Vector Quantizer Encoder
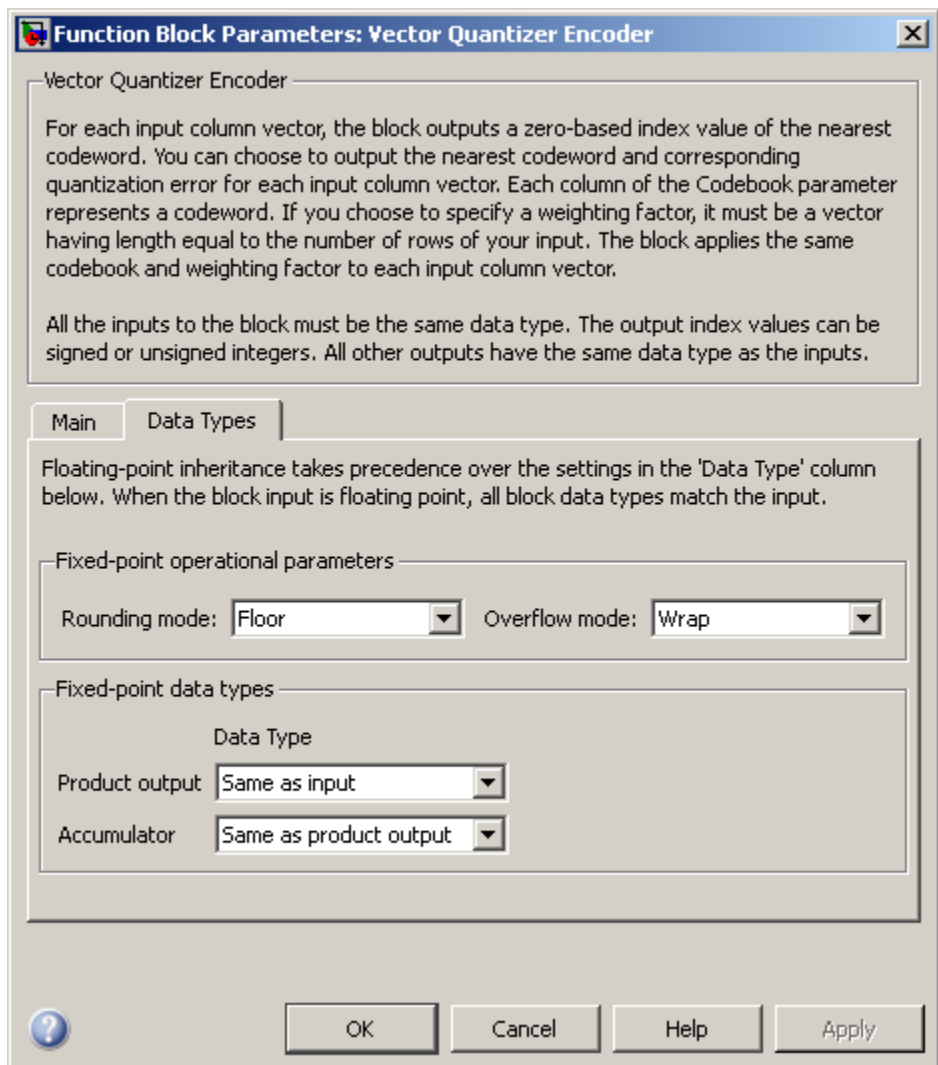
Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode to be used when block inputs are fixed point.

Product output

As depicted above, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how you would like to designate this product output word and fraction lengths.

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

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

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

Accumulator
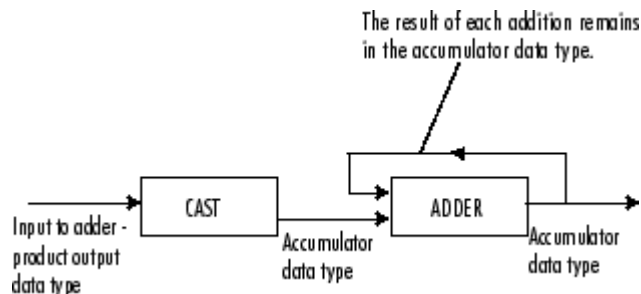
As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the

accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate the accumulator word and fraction lengths.

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

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

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

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

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression.* Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

Port	Supported Data Types
U	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
C	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
W	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
I	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Q(U)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
D	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	DSP System Toolbox
Scalar Quantizer Design	DSP System Toolbox
Uniform Encoder	DSP System Toolbox
Uniform Decoder	DSP System Toolbox
Vector Quantizer Decoder	DSP System Toolbox

Vector Scope

Purpose
Display vector or matrix of time-domain, frequency-domain, or user-defined data

Library
Sinks

dspsnks4

Description
The Vector Scope block is a comprehensive display tool similar to a digital oscilloscope. The block can display time-domain, frequency-domain, or user-defined signals. You can use the Vector Scope block to plot consecutive time samples from a vector, or to plot vectors containing data such as filter coefficients or spectral magnitudes. To compute and plot the periodogram of a signal with a single block, use the Spectrum Scope block.

The input to the Vector Scope block can be any real-valued matrix or vector. The block treats each column of an M-by-N matrix input as an independent channel of data with M consecutive samples.

The block plots each sample of each input channel sequentially across the horizontal axis of the plot.

Scope Properties Pane

The **Scope Properties** pane enables you to plot time-domain, frequency-domain, or user-defined data, and adjust the horizontal display span of the plot. The scope displays frames of data, and updates the display for each new input frame.

The **Input domain** parameter specifies the domain of the input data. If you select Time, for M-by-N inputs containing time-domain data, the block treats each of the N input frames (columns) as a succession of M consecutive samples taken from a time series. That is, each data point in the input frame is assumed to correspond to a unique time value. Also, the **Time display span (number of frames)** parameter appears on the pane. Enter a scalar value greater than or equal to one that corresponds to the number of frames to be displayed across the width of the scope window.

If you select Frequency for the **Input domain** parameter, for *M*-by-*N* inputs containing frequency-domain data, the block treats each of the *N* input frames (columns) as a vector of spectral magnitude data corresponding to *M* consecutive ascending frequency indices. That is, when the input is a single column vector, u, each value in the input frame, u(i), is assumed to correspond to a unique frequency value, f(i), where f(i+1)>f(i).

If you select User-defined for the **Input domain** parameter, the block does not assume that the input frame data is time-domain or frequency-domain data. You can plot the data in the appropriate manner. Also, the **Horizontal display span (number of frames)** parameter appears on the pane. Enter a scalar value greater than or equal to one that corresponds to the number of frames to be displayed across the width of the scope window.
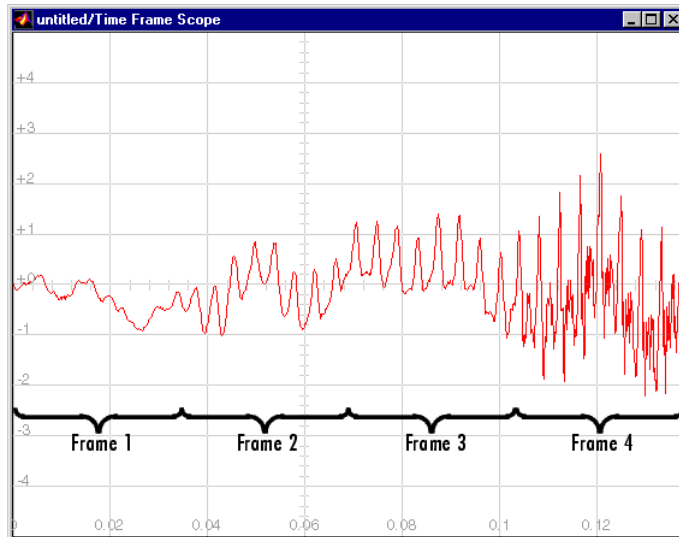
Display Properties Pane

The **Display Properties** pane enables you to control how the block displays your data.

Vector Scope

The **Show grid** parameter toggles the background grid on and off.

If you select the **Persistence** check box, the window maintains successive displays. That is, the scope does not erase the display after each frame (or collection of frames), but overlays successive input frames in the scope display.

If you select the **Frame number** check box, the block displays the number of the current frame in the input sequence on the scope window, and the block increments the count as each new input is received. Counting starts at 1 with the first input frame, and continues until the simulation stops.

If you select the **Channel legend** check box, a legend indicating the line color, style, and marker of each channel's data is added. When the input signal is labeled, that label appears in the channel legend. When the input signal is not labeled, but comes from a Concatenate block or a Mux block with labeled inputs, those labels appear in the channel legend. Otherwise, each channel in the legend is labeled with the channel number (CH 1, CH 2, etc.). Click and drag the legend to reposition it in the scope window; double-click on the line label to edit the text. If you rerun the simulation, the labels revert to the defaults.

If you select the **Compact display** check box, the scope completely fills the figure window. The scope does not display menus and axis titles, but it does show the numerical axis labels within the axes. If you clear the **Compact display** check box, the scope displays the axis labels and titles in a gray border surrounding the scope axes, and the window's menus and toolbar become visible.

If you select the **Open scope at start of simulation** check box, the scope opens at the start of the simulation. If you clear this parameter, the scope does not open automatically during the simulation. You can use this feature when you have several scope blocks in a model, and you do not want to view all the associated scopes during the simulation.

> **Note** Before running a model that contains a Vector Scope block in Accelerator, Rapid Accelerator, or External mode, you must select the **Open scope at start of simulation** check box. If you do not select this check box before running your model for the first time, the scope will not display your simulation data.

If you want to view a scope window that is not open during simulation, click **Open scope immediately** on the **Display Properties** pane of the desired Scope block.

The **Scope position** parameter specifies a four-element vector of the form

```
[left bottom width height]
```

specifying the position of the scope window on the screen, where (0,0) is the lower-left corner of the display. See the MATLAB `figure` function for more information.

Axis Properties Pane

The parameters that are available on the **Axis Properties** pane depend on the setting of the **Input domain** parameter on the **Scope Properties** pane.

Time Domain Inputs

When **Time display limits** is set to Auto, the block scales the horizontal axis of time-domain signals automatically. The range of the

time axis is $[0, S*T_{fi}]$, where T_{fi} is the input frame period, and S is the **Time display span (number of frames)** parameter on the **Scope**

Properties pane. The spacing between time points is $T_{fi}/(M-1)$, where M is the number of samples in each consecutive input frame.

When **Time display limits** is set to User-defined, the **Minimum X-limit (s)** and **Maximum X-limit (s)** parameters set the range of the horizontal axis.

Minimum Y-limit and **Maximum Y-limit** parameters set the range of the vertical axis. Setting these parameters is analogous to setting the `ymin` and `ymax` values of the MATLAB axis function.

The **Y-axis label** is the text displayed to the left of the *y*-axis.

Frequency Domain Inputs

The **Frequency units** parameter specifies whether the frequency axis values should be in units of Hertz or rad/sec. When the **Frequency units** parameter is set to Hertz, the spacing between frequency points is $1/(M * T_s)$, where T_s is the sample time of the original time-domain signal. When the **Frequency units** parameter is set to rad/sec, the spacing between frequency points is $2\pi/(M * T_s)$.

The **Frequency range** parameter specifies the range of frequencies over which the magnitudes in the input should be plotted. The available options are [0..Fs/2], [-Fs/2..Fs/2], and [0..Fs], where Fs is the original time-domain signal's sample frequency. The Vector Scope block assumes that the input data spans the range $[0, F_s)$, which is the same as the output from an FFT. To plot over the range [0..Fs/2] the scope truncates the input vector, leaving only the first half of the data, then plots these remaining samples over half the frequency range. To plot over the range [-Fs/2..Fs/2], the scope reorders the input vector elements such that the last half of the data becomes the first half, and vice versa; then it relabels the *x*-axis accordingly.

If you select the **Inherit sample time from input** check box for frequency domain inputs, 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.

Vector Scope

- 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, specify the appropriate value for the **Sample time of original time series** parameter.

When **Frequency display limits** is set to Auto, the block scales the horizontal axis of frequency-domain signals automatically. To do this, the Vector Scope block needs to know the sample period of the original time-domain sequence represented by the frequency-domain data. Specify this period by entering a value for the **Sample time of original time series** parameter.

When **Frequency display limits** is set to User-defined, the **Minimum frequency** and **Maximum frequency** parameters set the range of the horizontal axis.

The **Y-axis scaling** parameter allows you to select Magnitude or dB scaling along the *y*-axis.

Minimum Y-limit and **Maximum Y-limit** parameters set the range of the vertical axis. Setting these parameters is analogous to setting the ymin and ymax values of the MATLAB axis function.

The **Y-axis label** is the text displayed to the left of the *y*-axis.

User-Defined Inputs

If you select the **Inherit sample increment from input** check box for user-defined input domains, the block scales the horizontal axis by computing the horizontal interval between samples in the input frame from the frame period of the input. For example, when the input frame period is 1, and there are 64 samples per input frame, the interval between samples is computed to be 1/64. Computing the interval this way is usually only valid when the following conditions hold:

- The input is a nonoverlapping time series; the *x*-axis on the scope represents time.

- The input sample period (1/64 in the above example) is equal to the period with which the physical signal was originally sampled.

In cases where not all of these conditions hold, use the **X display offset (samples)** and **Increment per sample in input** parameters.

The **X-axis title** is the text displayed below the *x*-axis.

When **X display limits** is set to Auto, the block scales the horizontal axis of user-defined domain signals automatically. To do this, the Vector Scope block needs to know the spacing of the input data. Specify this spacing using the **Increment per sample in input** parameter, I_s. This parameter represents the numerical interval between adjacent *x*-axis points corresponding to the input data. The range of the horizontal axis is $[0, M * I_s * S]$, where M is the number of samples in each consecutive input frame, and S is the **Horizontal display span (number of frames)** parameter that you specify in the **Scope Properties** pane.

When **X display limits** is set to User-defined, the **Minimum X-limit (samples)** and **Maximum X-limit (samples)** parameters set the range of the horizontal axis.

Minimum Y-limit and **Maximum Y-limit** parameters set the range of the vertical axis. Setting these parameters is analogous to setting the ymin and ymax values of the MATLAB axis function.

The **Y-axis label** is the text displayed to the left of the *y*-axis.

Line Properties Pane

Use the parameters on the **Line Properties** pane to help you distinguish between two or more independent channels of data on the scope.

The **Line visibilities** parameter specifies which channel's data is displayed on the scope, and which is hidden. The syntax specifies the visibilities in list form, where the term on or off as a list entry specifies the visibility of the corresponding channel's data. The list entries are separated by the pipe symbol, |.

For example, a five-channel signal would ordinarily generate five distinct plots on the scope. To disable plotting of the third and fifth lines, enter the following visibility specification in the **Line visibilities** parameter.

```
 on | on | off | on | off
ch 1  ch 2  ch 3   ch 4   ch 5
```

Note that the first (leftmost) list item corresponds to the first signal channel (leftmost column of the input matrix).

The **Line styles** parameter specifies the line style with which each channel's data is displayed on the scope. The syntax specifies the channel line styles in list form, with each list entry specifying a style for the corresponding channel's data. The list entries are separated by the pipe symbol, |.

For example, a five-channel signal would ordinarily generate all five plots with a solid line style. To plot each line with a different style, enter

```
  - | -- | : | -. | -
ch 1  ch 2  ch 3 ch 4  ch 5
```

These settings plot the signal channels with the following styles.

Line Style	Command to Type in Line Style Parameter	Appearance
Solid	-	————————
Dashed	- -	— — — — — — —
Dotted	:	····························
Dash-dot	- .	— · — · — · — · —
No line	none	No line appears

Note that the first (leftmost) list item, '-', corresponds to the first signal channel (leftmost column of the input matrix). See the LineStyle property of the MATLAB `line` function for more information about the style syntax.

The **Line markers** parameter specifies the marker style with which each channel's samples are represented on the scope. The syntax specifies the channels' marker styles in list form, with each list entry specifying a marker for the corresponding channel's data. The list entries are separated by the pipe symbol, |.

For example, a five-channel signal would ordinarily generate all five plots with no marker symbol (that is, the individual sample points are not marked on the scope). To instead plot each line with a different marker style, you could enter

```
 *  |  .  |  x  |  s  |  d
ch 1   ch 2   ch 3   ch 4   ch 5
```

These settings plot the signal channels with the following styles.

Marker Style	Command to Type in Marker Style Parameter	Appearance
Asterisk	*	
Point	.	
Cross	x	

Marker Style	Command to Type in Marker Style Parameter	Appearance
Square	s	
Diamond	d	

Note that the leftmost list item, `'*'`, corresponds to the first signal channel or leftmost column of the input matrix. See the LineSpec property of the MATLAB `line` function for more information about the available markers.

To produce a stem plot for the data in a particular channel, type the word `stem` instead of one of the basic marker shapes.

The **Line colors** parameter specifies the color in which each channel's data is displayed on the scope. The syntax specifies the channel colors in list form, with each list entry specifying a color (in one of the MATLAB `ColorSpec` formats) for the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate all five plots in the color black. To instead plot the lines with the color order below, enter

```
[0 0 0] | [0 0 1] | [1 0 0 ] | [0 1 0] | [.7529 0 .7529]
 ch 1       ch 2       ch 3       ch 4         ch 5
```

or

```
'k' | 'b' | 'r' | 'g' | [.7529 0 .7529]
ch 1   ch 2   ch 3   ch 4        ch 5
```

These settings plot the signal channels in the following colors (8-bit RGB equivalents shown in the center column).

Color	RGB Equivalent	Appearance
Black	(0,0,0)	
Blue	(0,0,255)	
Red	(255,0,0)	
Green	(0,255,0)	
Dark purple	(192,0,192)	

Note that the leftmost list item, 'k', corresponds to the first signal channel or leftmost column of the input matrix. See the MATLAB function ColorSpec for more information about the color syntax.

Vector Scope Window

The title that appears in the title bar of the scope window is the same as the block title. In addition to the standard MATLAB figure window menus such as **File**, **Window**, and **Help**, the Vector Scope window contains **View**, **Axes**, and **Channels** menus.

The options in the **View** menu allow you to zoom in and out of the scope window:

- To zoom in on the scope window, you must first select **View > Zoom In** or click the corresponding Zoom In toolbar button (🔍). You can then zoom in by clicking in the center of your area of interest, or by clicking and dragging your cursor to draw a rectangular area of interest inside of the scope window.

- To zoom in on the *x*-axis of the scope window, you must first select **View > Zoom X**, or click the corresponding Zoom X-Axis toolbar

button (⊗) on the scope window. You can then zoom in on the
x-axis with a single click inside the scope window or by clicking and
dragging the cursor along the *x*-axis over your area of interest.

- To zoom in on the *y*-axis of the scope window, you must first select
  View > Zoom Y or click the corresponding Zoom Y-Axis toolbar

  button (⊘). You can then zoom in on the *y*-axis with a single click
  inside the scope window or by clicking and dragging the cursor along
  the *y*-axis over your area of interest.

- To return to the original view of the scope window, you have the
  following options:

  - Select **Full View** from the **View** menu .

  - Click the Restore default view toolbar button (✕) on the
    Vector Scope window.

  - Right-click inside the scope window, and select Reset to
    Original View.

Note To zoom out in smaller increments, you can right-click inside
of the scope window and select Zoom Out. You can also zoom out by
holding down the **Shift** key and clicking the left mouse button inside
the scope window.

The parameters that you set using the **Axes** menu apply to all channels.
Many of the parameters in this menu are also accessible through the
block parameters dialog box. For descriptions of these parameters, see
"Display Properties Pane" on page 2-1731. Below are descriptions of
other parameters in the **Axes** menu:

- **Refresh** erases all data on the scope display, except for the most
  recent trace. This command is useful in conjunction with the
  Persistence setting.

- **Autoscale** resizes the *y*-axis to best fit the vertical range of the data.

> **Note** The **Minimum Y-limit** and **Maximum Y-limit** parameters
> on the **Axis properties** pane of the block dialog are not updated to
> display the numerical limits selected by the autoscale feature.

- **Save Axes Settings** allows you to save the current axes settings.
  When you select this option, the **Minimum Y-limit** and **Maximum
  Y-limit** parameters of the **Axes Properties** pane update with the
  current *y*-axes limits. The **Time display limits** (or **Frequency
  display limits**) parameter is set to User-defined, and the current
  x-axes limits are saved in the **Minimum X-limit** and **Maximum
  X-limit** (or **Minimum Frequency** and **Maximum Frequency**)
  parameters. To save these axes settings for your next MATLAB
  session, you need to resave your model.

- **Save Scope Position** updates the **Scope position** parameter on
  the **Display Properties** pane of the block dialog to reflect the scope
  window's current position and size. To make the scope window open
  at a particular location on the screen when the simulation runs, drag
  the window to the desired location, resize it, and select **Save Scope
  Position** from the **Axes** menu.

The properties listed in the **Channels** menu apply to a particular
channel. All of the parameters in this menu are also accessible through
the block parameters dialog box. For descriptions of these parameters,
see "Line Properties Pane" on page 2-1737.

Many of these options can also be accessed by right-clicking with the
mouse anywhere on the scope display. The menu that is displayed
contains a combination of the options available in the **View**, **Axes** and
Channels menus.

Vector Scope

Dialog Box

Scope Properties Pane

Input domain

Select the domain of the input. Your choices are Time, Frequency, or User-defined. Tunable.

Time display span (number of frames)

The number of consecutive frames to display (horizontally) on the scope at any one time. This parameter is visible when the **Input domain** parameter is set to `Time`.

Horizontal display span (number of frames)

The number of consecutive frames to display (horizontally) on the scope at any one time. This parameter is visible when the **Input domain** parameter is set to `User-defined`.

Display Properties Pane

Show grid

Toggle the scope grid on and off.Tunable.

Persistence

Select this check box to maintain successive displays. That is, the scope does not erase the display after each frame (or collection

of frames), but overlays successive input frames in the scope display. Tunable.

Frame number

If you select this check box, the number of the current frame in the input sequence appears in the Vector Scope window. Tunable.

Channel legend

Toggles the legend on and off. Tunable.

Compact display

Resizes the scope to fill the window. Tunable.

Open scope at start of simulation

Select this check box to open the scope at the start of the simulation. When this parameter is cleared, the scope does not open automatically during the simulation. Tunable.

Note Before running a model that contains a Vector Scope block in Accelerator, Rapid Accelerator, or External mode, you must select the **Open scope at start of simulation** check box. If you do not select this check box before running your model for the first time, the scope will not display your simulation data.

Open scope immediately

If the scope is not open during simulation, select this check box to open it. This parameter is visible only while the simulation is running.

Scope position

A four-element vector of the form [left bottom width height] specifying the position of the scope window. (0,0) is the lower-left corner of the display. Tunable.

Axis Properties Pane

The parameters that are available on the **Axis Properties** pane depend on the setting of the **Input domain** parameter on the **Scope**

Properties pane. When `Time` is selected for the **Input domain** parameter, the following parameters are available on the **Axis Properties** pane:
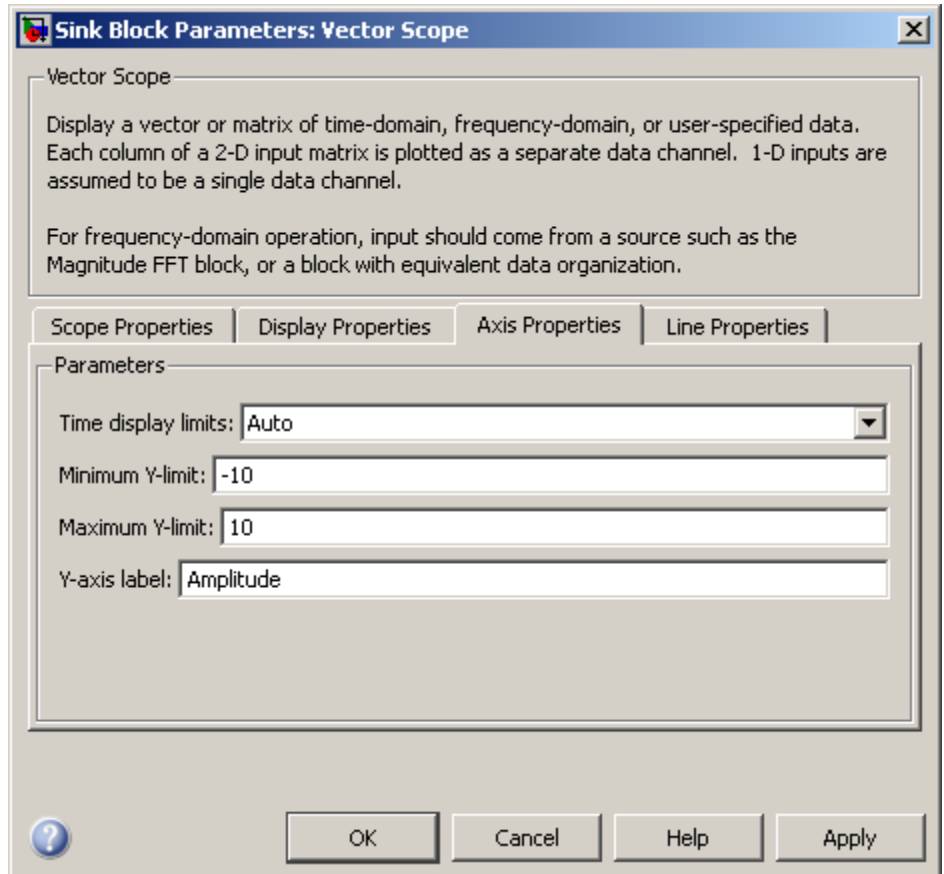
Time display limits

Select `Auto` to have the limits of the *x*-axis set for you automatically, or `User-defined` to set the limits yourself in the **Minimum X-limit (s)** and **Maximum X-limit (s)** parameters.

Minimum X-limit (s)

> Specify the minimum value of the *x*-axis in seconds. This parameter is only visible if the **Time display limits** parameter is set to `User-defined`. Tunable.

Maximum X-limit (s)

> Specify the maximum value of the *x*-axis in seconds. This parameter is only visible if the **Time display limits** parameter is set to `User-defined`. Tunable.

Minimum Y-limit

> Specify the minimum value of the *y*-axis. Setting this parameter is analogous to setting the `ymin` value of the MATLAB `axis` function. Tunable.

Maximum Y-limit

> Specify the maximum value of the *y*-axis. Setting this parameter is analogous to setting the `ymax` value of the MATLAB `axis` function. Tunable.

Y-axis label

> Specify text to be displayed to the left of the *y*-axis. Tunable.

When `Frequency` is selected for the **Input domain** parameter, the following parameters are available on the **Axis Properties** pane:
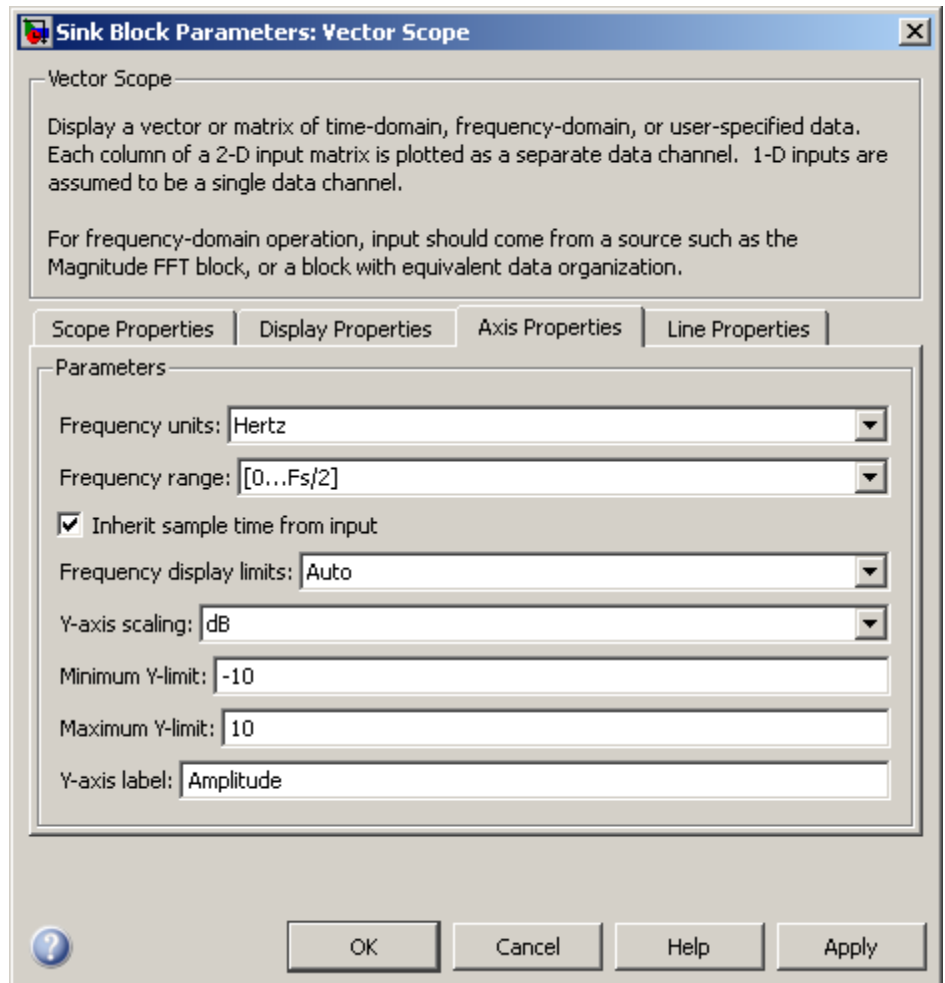
Frequency units

Choose the frequency units for the *x*-axis, Hertz or rad/sec.
Tunable.

Vector Scope

Frequency range

    Specify the frequency range over which to plot the data. Tunable.

Inherit sample time from input

    If you select this check box, the block computes the time-domain sample period from the frame period and frame size of the frequency-domain input. Use this parameter only when the length of the each frame of frequency-domain data is the same as the length of the frame of time-domain data from which it was generated. Tunable.

Sample time of original time series

    Enter the sample period, T_s, of the original time-domain signal. This parameter is only visible when the **Inherit sample time from input** check box is not selected. Tunable.

Frequency display limits

    Select Auto to have the limits of the *x*-axis set for you automatically, or User-defined to set the limits yourself in the **Minimum frequency** and **Maximum frequency** parameters.

Minimum frequency

    Specify the minimum frequency value of the *x*-axis in Hertz or rad/sec. This parameter is only visible if the **Frequency display limits** parameter is set to User-defined. Tunable.

Maximum frequency

    Specify the maximum frequency value of the *x*-axis in Hertz or rad/sec. This parameter is only visible if the **Frequency display limits** parameter is set to User-defined. Tunable.

Y-axis scaling

    Choose either dB (decibel) or Magnitude scaling for the *y*-axis. Tunable.

Minimum Y-limit

    Specify the minimum value of the *y*-axis. Setting this parameter is analogous to setting the ymin value of the MATLAB axis function. Tunable.

Maximum Y-limit

Specify the maximum value of the *y*-axis. Setting this parameter is analogous to setting the `ymax` value of the MATLAB `axis` function. Tunable.

Y-axis label

Specify text to be displayed to the left of the *y*-axis. Tunable.

When `User-defined` is selected for the **Input domain** parameter, the following parameters are available on the **Axis Properties** pane:

Vector Scope
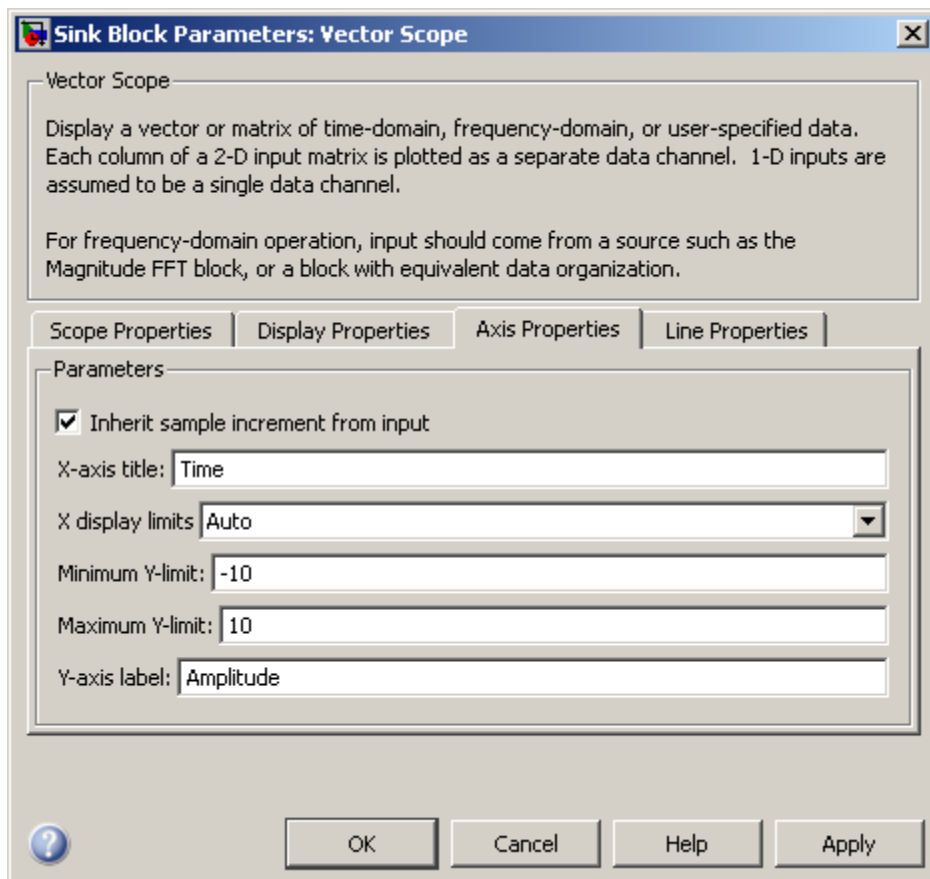
Inherit sample increment from input

> When you select this check box, the block scales the horizontal
> axis by computing the horizontal interval between samples in
> the input frame from the frame period of the input. Use this
> parameter only when the input's sample period is equal to the
> period with which the physical signal was originally sampled.
> Tunable.

X display offset (samples)

Specify an offset for the *x*-axis display in samples. This parameter is only visible when the **Inherit sample increment from input** check box is not selected. Tunable.

Increment per sample in input

Enter the numerical interval between adjacent *x*-axis points corresponding to the user-defined input data. This parameter is only visible when the **Inherit sample increment from input** check box is not selected. Tunable.

X-axis title

Enter the text to be displayed below the *x*-axis. Tunable.

X display limits

Select `Auto` to have the limits of the *x*-axis set for you automatically, or `User-defined` to set the limits yourself in the **Minimum X-limit (samples)** and **Maximum X-limit (samples)** parameters.

Minimum X-limit (samples)

Specify the minimum value of the *x*-axis in samples. This parameter is only visible if the **X display limits** parameter is set to `User-defined`. Tunable.

Maximum X-limit (samples)

Specify the maximum value of the *x*-axis in samples. This parameter is only visible if the **X display limits** parameter is set to `User-defined`. Tunable.

Minimum Y-limit

Specify the minimum value of the *y*-axis. Setting this parameter is analogous to setting the `ymin` value of the MATLAB `axis` function. Tunable.

Maximum Y-limit

Specify the maximum value of the *y*-axis. Setting this parameter is analogous to setting the `ymax` value of the MATLAB `axis` function. Tunable.

Y-axis label
> Specify text to be displayed to the left of the *y*-axis. Tunable.

Line Properties Pane
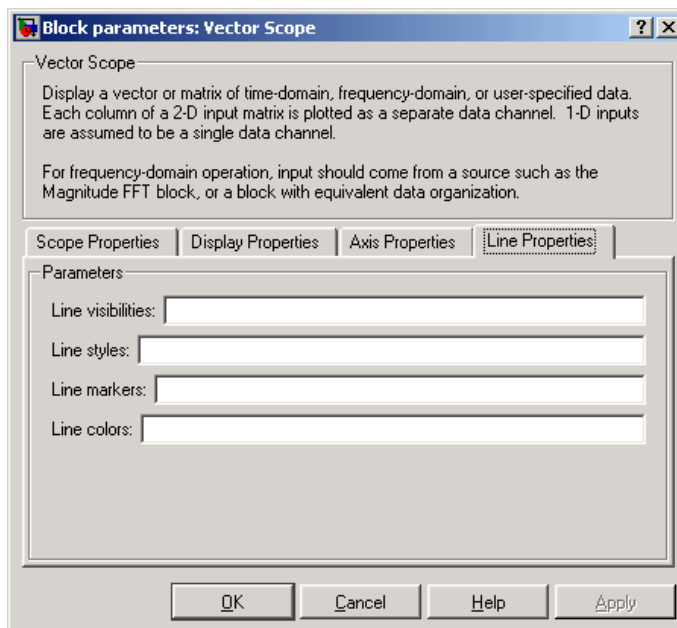
Line visibilities
> Enter on or off to specify the visibility of the various channels' scope traces. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Line styles
> Enter the line styles of the various channels' scope traces. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Line markers

Enter the line markers of the various channels' scope traces. Separate your choices for each channel with by a pipe (|) symbol.Tunable.

Line colors

Enter the colors of the various channels' scope traces using the ColorSpec formats. Separate your choices for each channel with by a pipe (|) symbol.Tunable.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

See Also

Matrix Viewer	DSP System Toolbox
Spectrum Scope	DSP System Toolbox

Waterfall

Purpose   View vectors of data over time

Library   Sinks

dspsnks4

Description   The Waterfall block displays multiple vectors of data at one time. These vectors represent the input data at consecutive sample times. The input to the block can be real or complex-valued data vectors of any data type including fixed-point data types. However, the input is converted to double-precision before the block processes the data. The Waterfall block displays only real-valued, double-precision vectors of data.

The data is displayed in a three-dimensional axis in the Waterfall window. By default, the *x*-axis represents amplitude, the *y*-axis represents samples, and the *z*-axis represents time. You can adjust the number of sample vectors that the block displays, move and resize the Waterfall window, and modify block parameter values during the simulation. The Waterfall window has toolbar buttons that enable you to zoom in on displayed data, suspend data capture, freeze the scope's display, save the scope position, and export data to the workspace. The toolbar buttons are labeled in the following figure, which shows the Waterfall window as it appears when you double-click a Waterfall block.
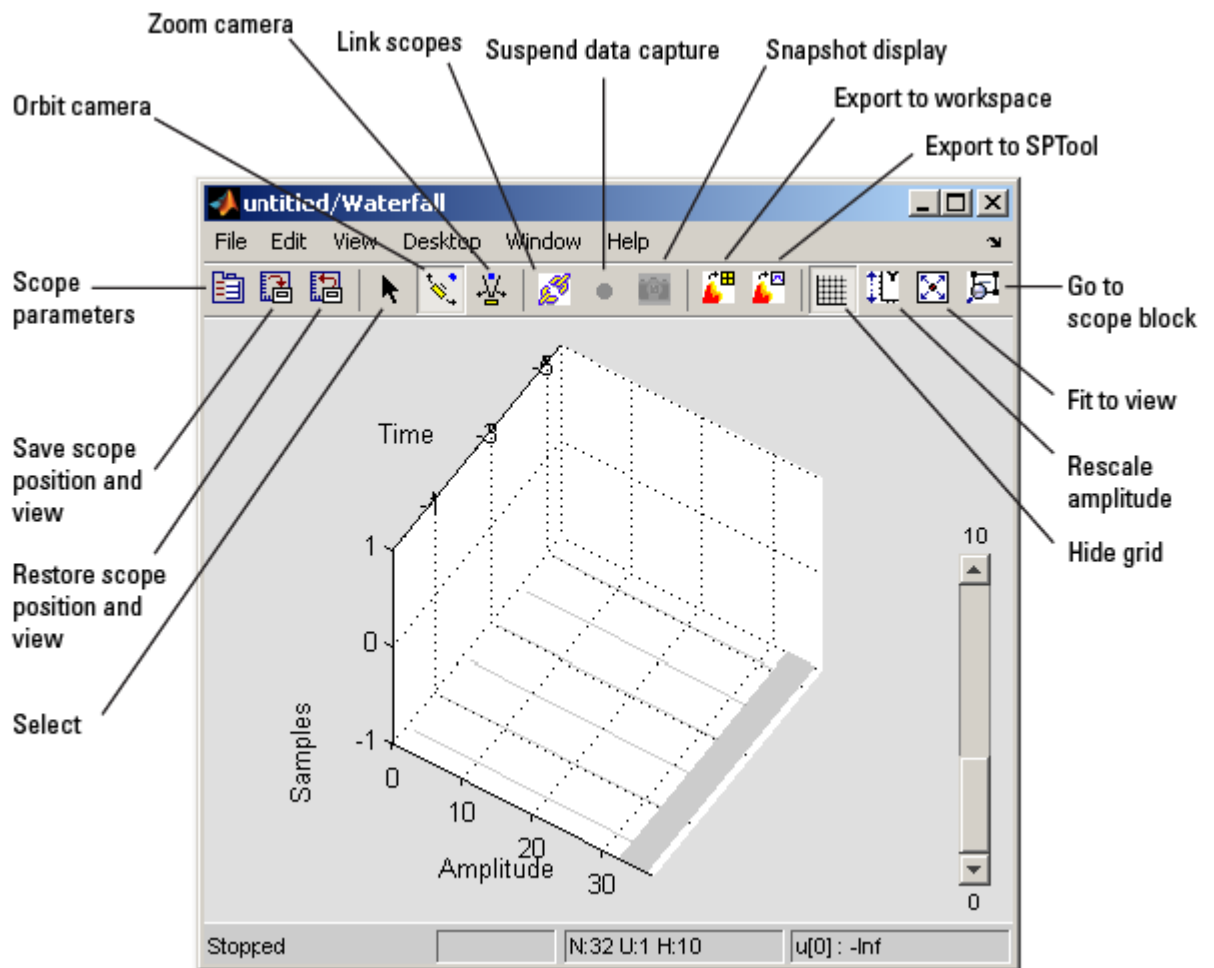
Sections of This Reference Page

- "Waterfall Parameters" on page 2-1758
- "Display Parameters" on page 2-1760
- "Axes Parameters" on page 2-1761

Waterfall

Waterfall Parameters

You can control the display and behavior of the Waterfall window using the Parameters dialog box.

Note You can alter the Waterfall parameters while the simulation is running. However, when you make changes to values in text boxes, you must click **Enter** or click outside the text box before the block accepts your changes.

---

1 To open the Parameters dialog box, click the **Scope parameters** button.
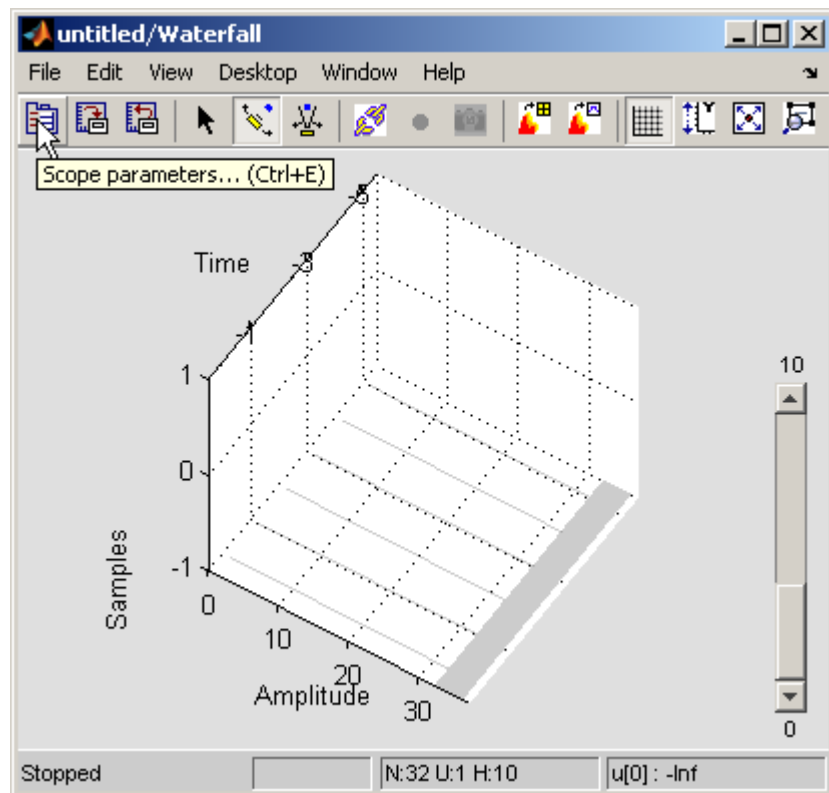
The Parameters dialog box appears.

Waterfall
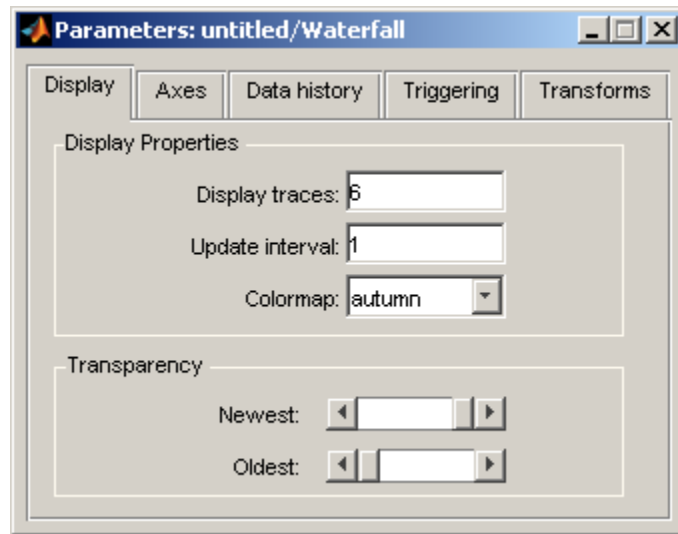
2 Click on the different panes to enter parameter settings.

Display Parameters

The following parameters control the Waterfall window's display.
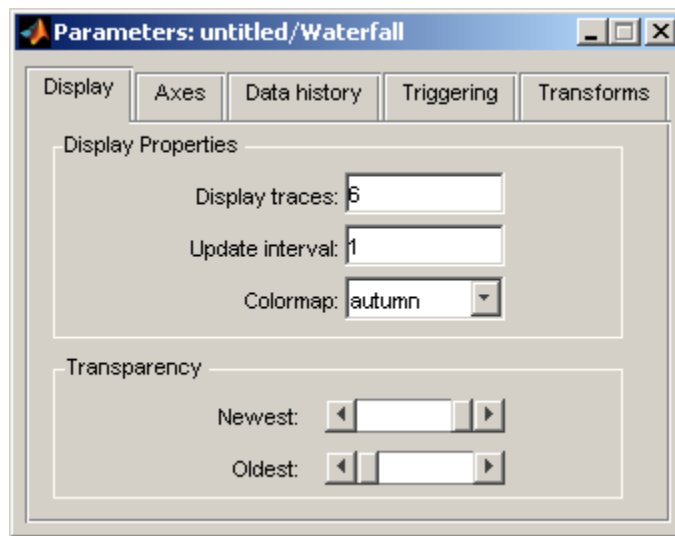
Display traces
> Enter the number of vectors of data to be displayed in the Waterfall window.

Update interval
> Enter the number of vectors the block should store before it displays them to the window.

Colormap
> Choose a colormap for the displayed data.

Transparency
> Specify the transparency of the newest and oldest data vectors. Placing the slider in the left-most position tells the block to make the data vector transparent. Placing the slider in the right-most position tells the block to make the data vector opaque. The intermediate data vectors transition between the two chosen transparency values.

Axes Parameters

The following parameters control the axes in the Waterfall window.

Waterfall
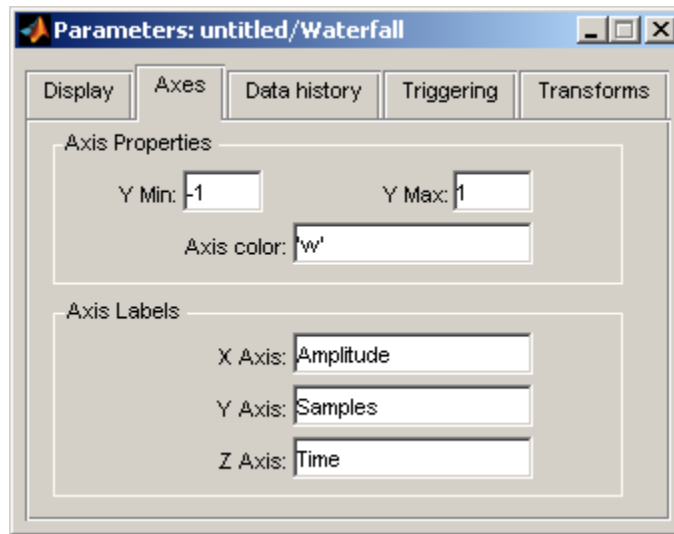
Y Min
> Enter the minimum value of the *y*-axis.

Y Max
> Enter the maximum value of the *y*-axis.

Axis color
> Enter a background color for the axes. Specify the color using a character string. For example, to specify black, enter `'k'`.

X Axis
> Enter the *x*-axis label.

Y Axis
> Enter the *y*-axis label.

Z Axis
> Enter the *z*-axis label.

Data History Parameters

The following parameters control how many input data vectors the Waterfall block stores. They also control how the data is exported to the MATLAB workspace or SPTool.
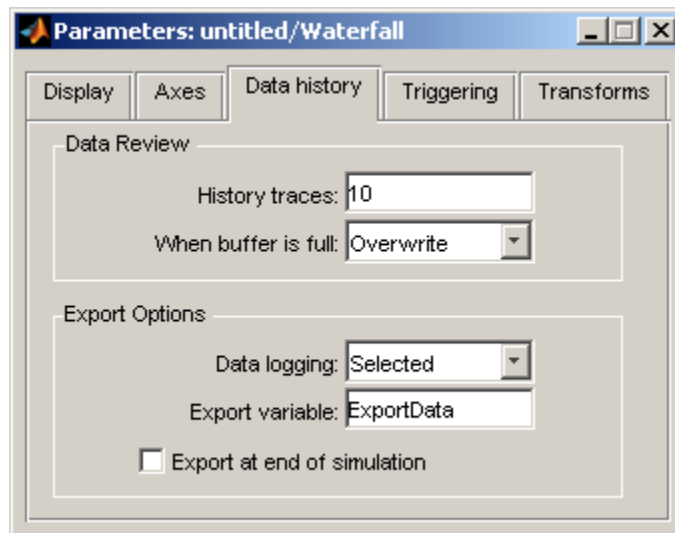
History traces

> Enter the number of vectors (traces) that you want the block to store.

When the buffer is full

> Use this parameter to control the behavior of the block when the buffer is filled:
>
> - Overwrite — The old data is replaced with the new data.
>
> - Suspend — The block stops storing data in the buffer; however, the simulation continues to run.
>
> - Extend — The block extends the buffer so that it can continue to store all the input data.

Data logging

Use this parameter to control which data is exported from the block:

- `Selected` — The selected data vector is exported.

- `All visible` — All of the data vectors displayed in the Waterfall window are exported.

- `All history` — All of the data vectors stored in the block's history buffer are exported.

Export variable

Enter the name of the variable that represents your data in the MATLAB workspace or SPTool. The default variable name is `ExportData`.

Export at end of simulation

Select this check box to automatically export the data to the MATLAB workspace when the simulation stops.

Triggering Parameters

The following parameters control when the Waterfall block starts and stops capturing data.
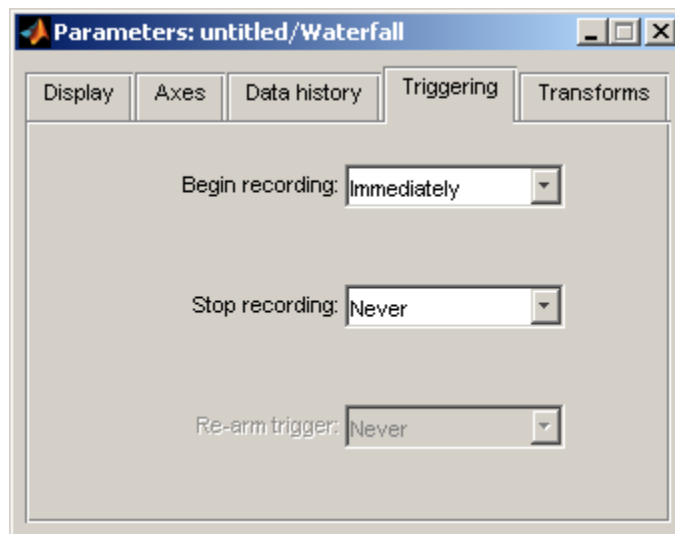
Begin recording

This parameter controls when the Waterfall block starts capturing data:

- `Immediately` — The Waterfall window captures the input data as soon as the simulation starts.

- `After T seconds` — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it begins capturing data.

- `After N inputs` — The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it begins capturing data.

- `User-defined` — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should begin capturing data. For more information about how you define this function, see "Scope Trigger Function" on page 2-1768.

Stop recording

> This parameter controls when the Waterfall block stops capturing data:
>
> - `Never` — The block captures the input data as long as the simulation is running.
>
> - `After T seconds` — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it stops capturing data.
>
> - `After N inputs` —The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it stops capturing data.
>
> - `User-defined` — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should stop capturing data. For more information about how you define this function, see "Scope Trigger Function" on page 2-1768.

Re-arm trigger

> This parameter controls when the Waterfall block begins waiting to capture data. It is available only when you select `After T seconds`, `After N inputs`, or `User-defined` for the **Stop recording** parameter:
>
> - `Never` — The Waterfall Scope block starts and stops capturing data as defined by the **Begin recording** and **Stop recording** parameters.
>
> - `After T seconds` — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it begins waiting to capture data.
>
> - `After N inputs` — The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it begins waiting to capture data.
>
> - `User-defined` — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that

defines when the block should begin waiting to capture data. For more information about how you define this function, see "Scope Trigger Function" on page 2-1768.

The triggering process is illustrated in the state diagram below.

Note: User-defined indicates the use of a scope trigger function.

Waterfall
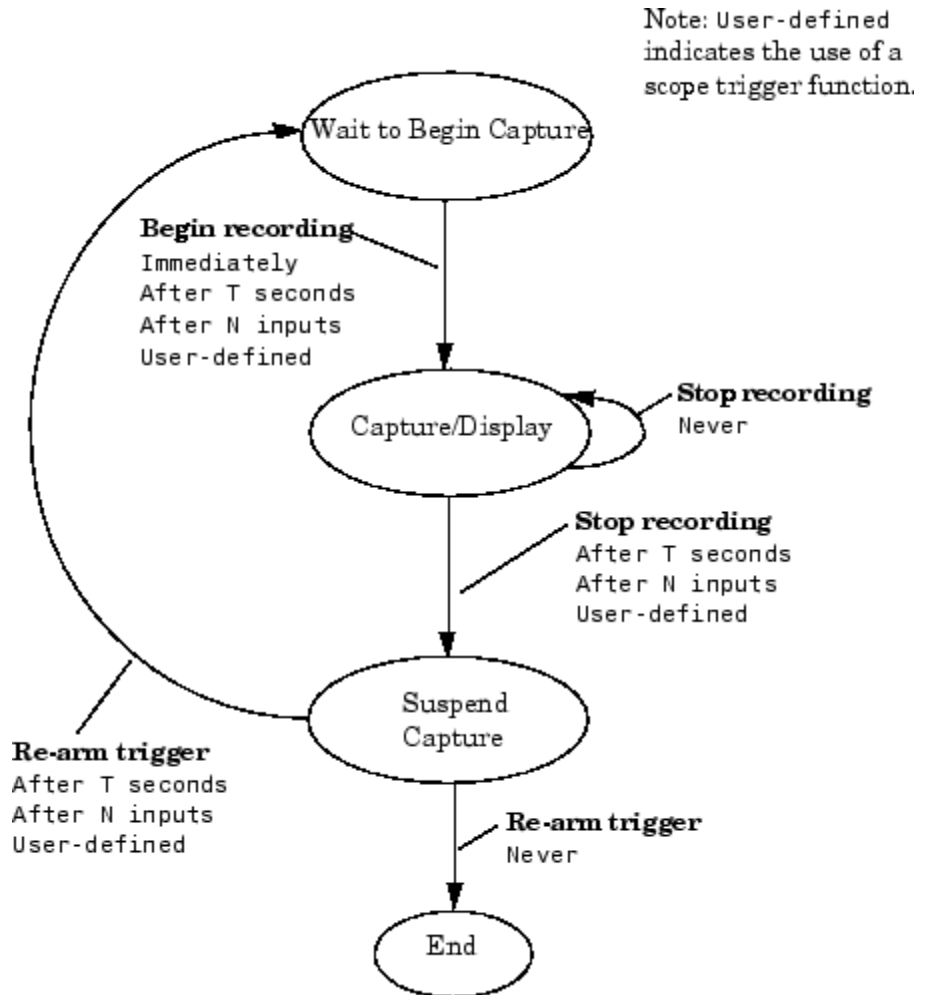
Scope Trigger Function

You can create custom scope trigger functions to control when the scope starts, stops, or begins waiting to capture data.
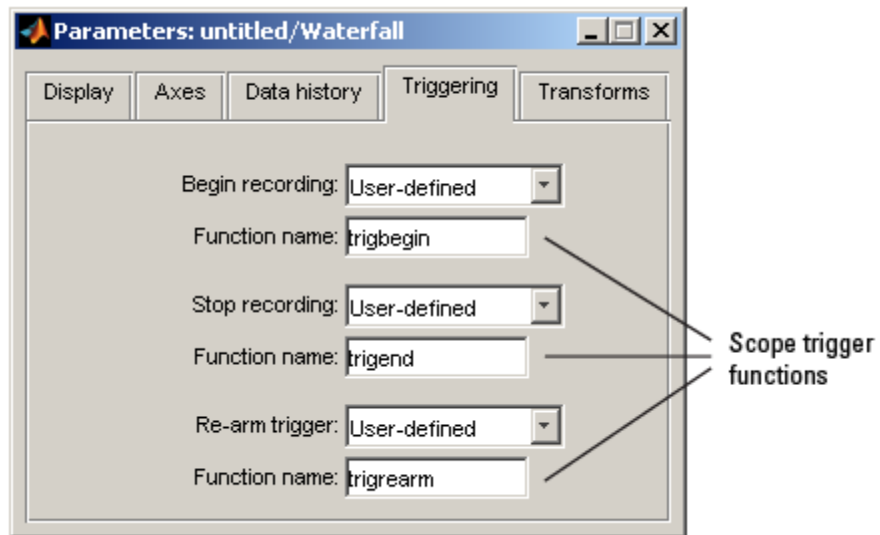
These functions must be valid MATLAB functions and be located either in the current folder or on the MATLAB path.

Each scope trigger function must have the following form

```
y = functionname(blk,t,u),
```

where `functionname` refers to the name you give your scope trigger function. The variable `blk` is the Simulink block handle. When the scope trigger function is called by the block, Simulink automatically populates this variable with the handle of the Waterfall block. The variable `t` is the current simulation time, represented by a real, double-precision, scalar value. The variable `u` is the vector input to the block. The output of the scope trigger function, `y`, is interpreted as a logical signal. It is either true or false:

- Begin recording scope trigger function

  - When the output of this scope trigger function is true, the Waterfall block starts capturing data.

  - When the output is false, the block remains in its current state.

- Stop recording scope trigger function

  - When the output of this scope trigger function is true, the block stops capturing data.

  - When the output is false, the block remains in its current state.

- Re-arm trigger scope trigger function

  - When the output of this scope trigger function is true, the block waits for a begin recording event.

  - When the output is false, the block remains in its current state.

Note The Waterfall block passes its input data directly to the scope trigger functions. These functions do not use the transformed data defined by the Transform parameters.

The following is an example of a scope trigger function. This function, called trigPower detects when the energy in u exceeds a certain threshold.

```
function y = trigPower(blk, t, u)

y = (u'*u > 2300);
```

The following is another example of a scope trigger function. This function, called count3, triggers the scope once three vectors with positive means are input to the block. Then, the function resets itself and begins searching for the next three input vectors with positive means. This scope trigger function is valid only when one Waterfall block is present in your model.

Waterfall

```
function y = count3(blk, t, u)

persistent state;
if isempty(state); state = 0; end
if mean(u)>0; state = state+1; end
y = (state>=3);
if y; state = 0; end
```

Transform Parameters

The following parameters transform the input data to the Waterfall
block. The result of the transform is displayed in the Waterfall window.
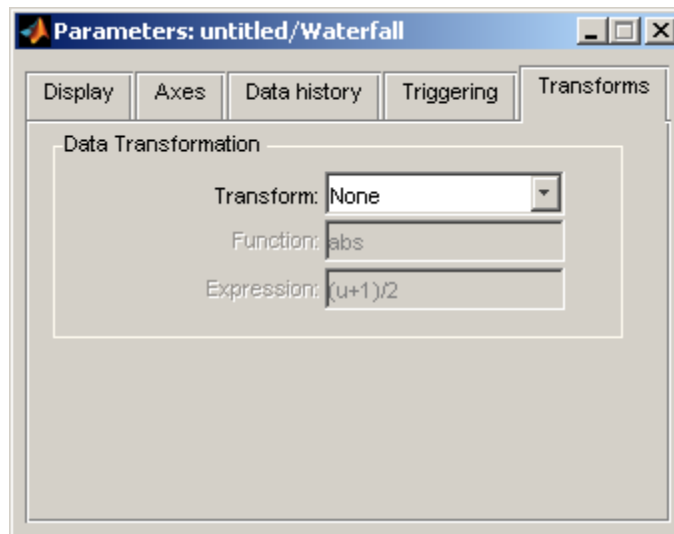
Note The block assumes that the input to the block corresponds to the **Transform** parameter you select. For example, when you choose `Complex-> Angle`, the block assumes that the input is complex. The block does not produce an error when the input is not complex. Therefore, you must verify the format of your input data to guarantee that a meaningful result is displayed in the Waterfall window.

Transform

Choose a transform that you would like to apply to the input of the Waterfall block:

- `None` — The input is displayed as it is received by the block.

- `Amplitude-> dB` — The block converts the input amplitude into decibels.

- `Complex-> Mag Lin` — The block converts the complex input into linear magnitude.

- `Complex-> Mag dB` — The block converts the complex input into magnitude in decibels.

- `Complex-> Angle` — The block converts the complex input into phase.

- `FFT-> Mag Lin Fs/2` — The block takes the linear magnitude of the FFT input and plots it from 0 to the Nyquist frequency.

- `FFT-> Mag dB Fs/2` — The block takes the magnitude of the FFT input, converts it to decibels, and plots it from 0 to the Nyquist frequency.

- `FFT-> Angle Fs/2` — The block converts the FFT input into phase and plots it from 0 to the Nyquist frequency.

- `Power-> dB` — The block converts the input power into decibels.

Waterfall

Function

This parameter is only available when you select `User-defined fcn` for the **Transform** parameter. Enter a function that you would like to apply to the input of the Waterfall block. For more information about how you define this function, see "Scope Transform Function" on page 2-1772.

Expression

This parameter is only available when you select `User-defined expr` for the **Transform** parameter. Enter an expression that you would like to apply to the input of the Waterfall block. The result of this expression must be real-valued. When you write the expression, be sure to include only one unknown variable. The block assumes this unknown variable represents the input to the block. When the block believes your expression is invalid, the following window appears.
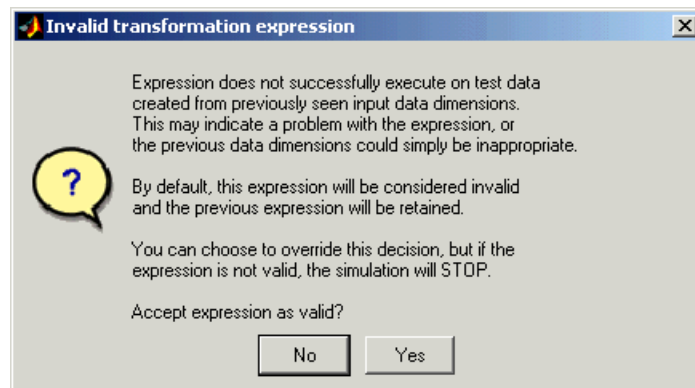
When you click **No**, your expression is not applied to the input. When you click **Yes** and your expression is invalid, your simulation stops and Simulink displays an error.

Scope Transform Function

You can create a scope transform function to control how the Waterfall block transforms your input data. This function must have a valid

MATLAB function name and be located either in the current folder or on the MATLAB path.

Your scope transform function must have the following form

```
y = functionname(u),
```

where `functionname` refers to the name you give your function. The variable `u` is the real or complex vector input to the block. The output of the scope transform function, `y`, must be a double-precision, real-valued vector. When it is not, the simulation stops and Simulink displays an error. Note that the output vector does not need to be the same size as the input vector.

Examples   The following examples illustrate some capabilities of the Waterfall block.

- "Exporting Data" on page 2-1773
- "Capturing Data" on page 2-1774
- "Linking Scopes" on page 2-1775
- "Selecting Data" on page 2-1776
- "Zooming" on page 2-1779
- "Rotating the Display" on page 2-1779
- "Scaling the Axes" on page 2-1779
- "Saving Scope Settings" on page 2-1780

Exporting Data

You can use the Waterfall block to export data to the MATLAB workspace or to SPTool:

1 Open and run the `dspanc` example.

2 While the simulation is running, click the **Export to Workspace** button.

3 Type `whos` at the MATLAB command line.

The variable `ExportData` appears in your MATLAB workspace. `ExportData` is a 40-by-6 matrix. This matrix represents the six data vectors that were present in the Waterfall window at the time you clicked the **Export to Workspace** button. Each column of this matrix contains 40 filter coefficients. The columns of data were captured at six consecutive instants in time.

You can control what data is exported using the **Data logging** parameter in **Data history** pane of the Parameters dialog box. For more information, see "Data History Parameters" on page 2-1763.

4 While the simulation is running, click the **Export to SPTool** button.

The SPTool GUI opens and the variable `ExportData` is displayed in the **Signals** list.

For more information about SPTool, see the Signal Processing Toolbox documentation.

Capturing Data

You can use the Waterfall block to interact with your data while it is being captured:

1 Open and run the `dspanc` example.

2 While the simulation is running, click the **Suspend data capture** button.

The Waterfall block no longer captures or displays the data coming from the Downsample block.

3 To continue capturing data, click the **Resume data capture** button.

4 To freeze the data display while continuing to capture data, click the **Snapshot display** button.

5 To view the Waterfall block that the data is coming from, click the **Go to scope block** button.

In the Simulink model window, the Waterfall block that corresponds to the active Waterfall window flashes. This feature is helpful when you have more than one Waterfall block in a model and you want to clarify which data is being displayed.

Linking Scopes

You can link several Waterfall blocks together in order to capture the effect of a model event in all of the Waterfall windows in the model:

1 Open the dspanc example.

2 Drag a second Waterfall block into the example model.

3 Connect this block to the Output port of the LMS Filter block as shown in the figure below.
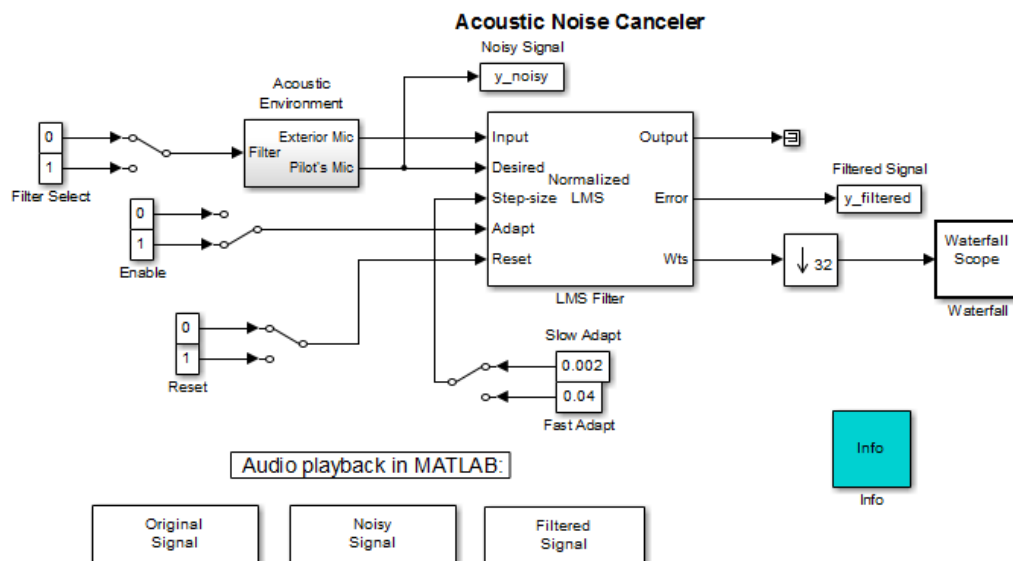
4 Run the model and view the model behavior in both Waterfall windows.

5 In the dspanc/Waterfall window, click the **Link scopes** button.

6 In the same window, click the **Suspend data capture** button.

The data capture is suspended in both scope windows.

7 Click the **Resume data capture** button.

The data capture resumes in both scope windows.

8 In the dspanc/Waterfall window, click the **Snapshot display** button.

In both scope windows, the data display freezes while the block continues to capture data.

9 To continue displaying the captured data, click the **Resume display** button.

Selecting Data

The following figure shows the Waterfall window displaying the output of the dspanc example:
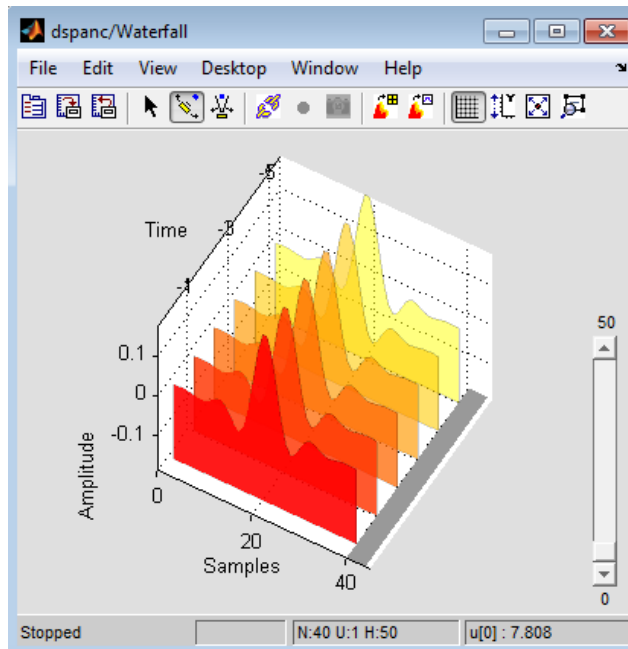
1 To select a particular set of data, click the **Select** button.

2 Click on the Time Bar at the bottom right of the axes to select a vector of data.

The Waterfall block highlights the selected trace.

Waterfall

Relative index Simulation time
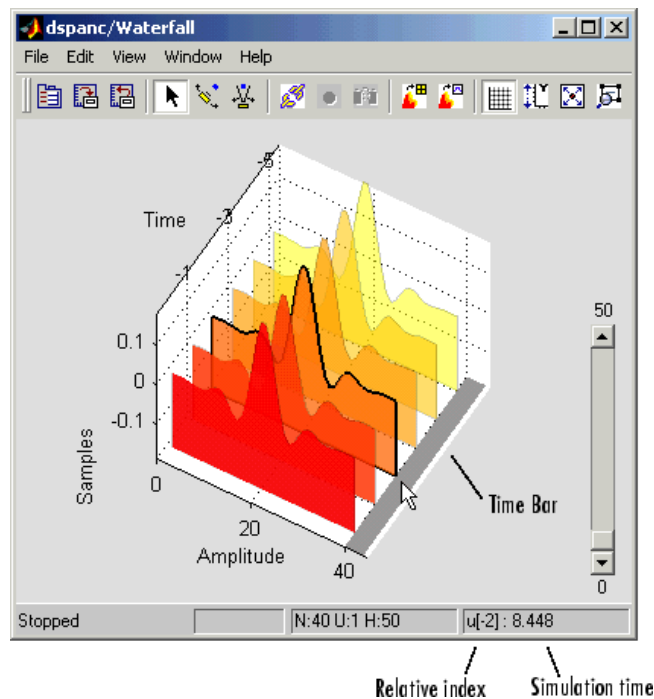
While the simulation is running, in the bottom right corner, the
Waterfall window displays the relative index of the selected trace.
For example, in the previous figure, the selected vector is two sample
times away from the most current data vector. When the simulation
is stopped, the Waterfall window displays both the relative index and
the simulation time associated with the selected trace.

3 To deselect the data vector, click it again.

4 Click-and-drag along the Time Bar.

Your selection follows the movement of the pointer.

You can use this feature to choose a particular vector to export to the MATLAB workspace or SPTool. For more information, see "Data History Parameters" on page 2-1763.

Zooming

You can use the Waterfall window to zoom in on data:

1 Click the **Zoom camera** button.

2 In the Waterfall window, click and hold down the left mouse button.

3 Move the mouse up and down and side-to-side to move closer and farther away from the axes.

4 To resize the axes to fit the Waterfall window, click the **Fit to view** button.

Rotating the Display

You can rotate the data displayed in the Waterfall window:

1 Click on the **Orbit camera** button.

2 In the Waterfall window, click and hold down the left mouse button.

3 Move the mouse in a circular motion to rotate the axes.

4 To return to the position of the original axes, click the **Restore scope position and view** button.

Scaling the Axes

You can use the Waterfall window to rescale the *y*-axis values:

1 Open and run the dspanc example.

2 Click the **Rescale amplitude** button.

The *y*-axis changes so that its minimum value is zero. The maximum value is scaled to fit the data displayed.

Waterfall

Alternatively, you can scale the *y*-axis using the **Y Min** and **Y Max** parameters in the **Axes** pane of the Parameters dialog box. This is helpful when you want to undo the effects of rescaling the amplitude. For more information, see "Axes Parameters" on page 2-1761.

Saving Scope Settings

The Waterfall block can save the screen position and viewpoint of the Waterfall window:

1 Click the **Save scope position and view** button.

2 Close the Waterfall window.

3 Reopen the Waterfall window.

   It reopens at the same place on your screen. The viewpoint of the axes also remains the same.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

The Waterfall block accepts any of these data types as input. However, the input is converted to double-precision before the block processes the data. The Waterfall block displays only real-valued, double-precision vectors of data.

See Also

Scope	Simulink
Time Scope	DSP System Toolbox

Vector Scope	DSP System Toolbox
Spectrum Scope	DSP System Toolbox
Matrix Viewer	DSP System Toolbox
Signal To Workspace	DSP System Toolbox
Triggered To Workspace	DSP System Toolbox

Wavelet Analysis (Obsolete)

Purpose   Decompose signal into components of logarithmically decreasing frequency intervals and sample rates (requires the Wavelet Toolbox product)

Library   dspobslib

Description
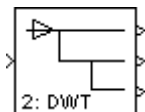
> **Note** The Wavelet Analysis block will be removed from the product in a future release. We strongly recommend replacing this block with the DWT block.

The Wavelet Analysis block uses the Wavelet Toolbox `wfilters` function 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
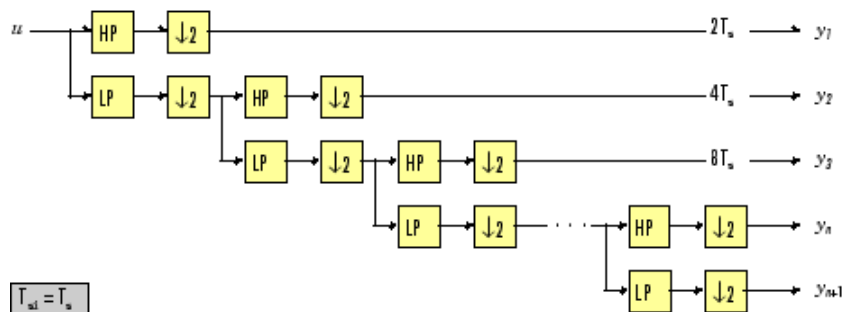
HP: highpass filter with $f_c \approx 1/2$ Nyquist
LP: lowpass filter with $f_c \approx 1/2$ Nyquist
↓2: downsample by 2

$T_{si} = T_s$

$T_{so} = (2^k)T_s$ for output y_k, $1 \leq k \leq n$
$T_{so} = (2^n)T_s$ for output y_{n+1}

At each level, the low-frequency output of the previous level is decomposed into adjacent high- and low-frequency subbands by a highpass (HP) and lowpass (LP) filter pair. Each of the two output subbands is half the bandwidth of the input to that level. The

2-1782

bandlimited output of each filter is maximally decimated by a factor of 2 to preserve the bit rate of the original signal.

Filter Coefficients

The filter coefficients for the highpass and lowpass filters are computed by the Wavelet Toolbox function wfilters, based on the wavelet specified in the **Wavelet name** parameter. The table below lists the available options.

Wavelet Name	Sample Wavelet Function Syntax
Haar	wfilters('haar')
Daubechies	wfilters('db4')
Symlets	wfilters('sym3')
Coiflets	wfilters('coif1')
Biorthogonal	wfilters('bior3.1')
Reverse Biorthogonal	wfilters('rbio3.1')
Discrete Meyer	wfilters('dmey')

The **Daubechies**, **Symlets**, and **Coiflets** options enable a secondary **Wavelet order** parameter that allows you to specify the wavelet order. For example, if you specify a **Daubechies** wavelet with **Wavelet order** equal to 6, the Wavelet Analysis block calls the wfilters function with input argument 'db6'.

The **Biorthogonal** and **Reverse Biorthogonal** options enable a secondary **Filter order [synthesis / analysis]** parameter that allows you to independently specify the wavelet order for the analysis and synthesis filter stages. For example, if you specify a **Biorthogonal** wavelet with **Filter order [synthesis / analysis]** equal to [2 / 6], the Wavelet Analysis block calls the wfilters function with input argument 'bior2.6'.

Wavelet Analysis (Obsolete)

See the Wavelet Toolbox documentation for more information about the `wfilters` function. If you want to explicitly specify the FIR coefficients for the analysis filter bank, use the Dyadic Analysis Filter Bank block.

Tree Structure

The wavelet tree structure has $n+1$ outputs, where n is the number of levels. The sample rate and bandwidth of the top output are half the input sample rate and bandwidth. The sample rate and bandwidth of each additional output (except the last) are half that of the output from the previous level. In general, for an input with sample period $T_{si} = T_s$, and bandwidth BW, output y_k has sample period $T_{so,k}$ and bandwidth BW_k.

$$
T_{so,k} = \begin{cases} \left(2^k\right)T_s & (1 \le k \le n) \\ \left(2^n\right)T_s & (k = n+1) \end{cases}
$$

$$
BW_k = \begin{cases} \dfrac{BW}{2^k} & (1 \le k \le n) \\ \dfrac{BW}{2^n} & (k = n+1) \end{cases}
$$

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.

Sample-Based Operation

An M-by-N sample-based matrix input is treated as $M*N$ independent channels, and the block filters each channel independently over time. The output at each port is the same size as the input, one output channel for each input channel. As described earlier, each output port has a different sample period.

The figure below shows the input and output sample periods for a 64-channel sample-based input to a three-level filter bank. The input has a period of 1, so the fastest output has a period of 2.
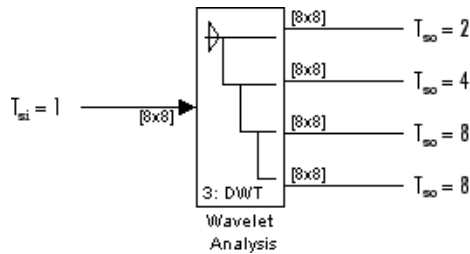
Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block filters each channel independently over time. The input frame size M_i must be a multiple of 2^n, and n is the number of filter bank levels. For example, a frame size of 8 would be appropriate for a three-level tree ($2^3=8$). The number of columns in each output is the same as the number of columns in the input.

Each output port has the same frame period as the input. The reduction in the output sample rates results from the smaller output frame sizes, as shown in the example below for a four-channel input to a three-level filter bank.

Wavelet Analysis (Obsolete)

$T_{fi} = 1$
$T_{si} = 1/64$
[64x4]

[32x4] $\quad T_{so} = 1/32)$

[16x4] $\quad T_{so} = 1/16)$

[8x4] $\quad T_{so} = 1/8)$

[8x4] $\quad T_{so} = 1/8)$

$T_{fo} = 1$

3: DWT
Wavelet
Analysis
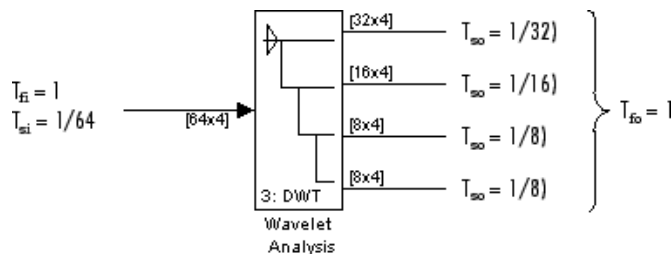
Zero Latency

The Wavelet Analysis block has no tasking latency for frame-based operation, which is always single-rate. The block therefore analyzes the first input sample (received at t=0) to produce the first output sample at each port.

Nonzero Latency

For sample-based operation, the Wavelet Analysis block is multirate and has 2^{n-1} samples of latency in both Simulink tasking modes. As a result, the block repeats a zero initial condition in each channel for the first 2^{n-1} output samples, before propagating the first analyzed input sample (computed from the input received at t=0).

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and the topic on models with multiple sample rates in the Simulink Coder documentation.

Dialog Box

Block Parameters: Wavelet Analysis

Wavelet Analysis (mask)

Multilevel discrete 1-D wavelet decomposition using a specific wavelet.

The Wavelet Toolbox must be installed to use this block.

Parameters

Wavelet name: Haar

Number of levels:

2

| OK | Cancel | Help | Apply |
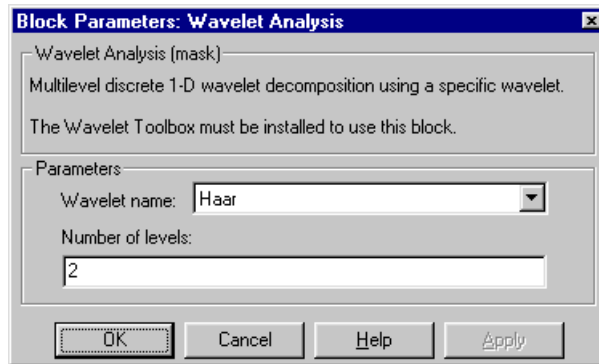
The parameters displayed in the dialog box vary for different wavelet types. Only some of the parameters listed below are visible in the dialog box at any one time.

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.

Wavelet Analysis (Obsolete)

References   Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets.* West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks.* Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks.* Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types   • Double-precision floating point

See Also

Dyadic Analysis Filter Bank	DSP System Toolbox
Wavelet Synthesis (Obsolete)	DSP System Toolbox
wfilters	Wavelet Toolbox

Purpose   Reconstruct signal from its multirate bandlimited components (requires the Wavelet Toolbox product)

Library   dspobslib

Description
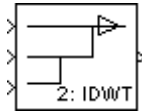
> **Note** The Wavelet Synthesis block will be removed from the product in a future release. We strongly recommend replacing this block with the IDWT block.

The Wavelet Synthesis block uses the Wavelet Toolbox wfilters function to reconstruct a signal that was decomposed by the Wavelet Analysis (Obsolete) 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
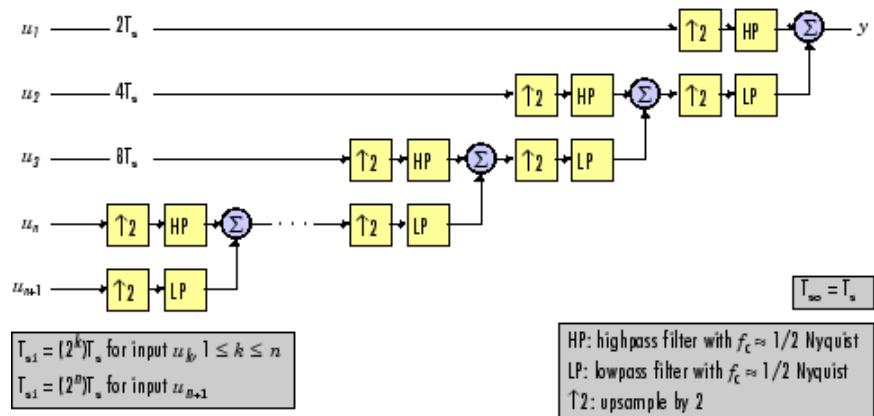
At each level, the two bandlimited inputs (one low-frequency, one high-frequency, both with the same sample rate) are upsampled by

Wavelet Synthesis (Obsolete)

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.

$$T_{si,k+1} = 2T_{si,k} \quad 1 \le k < n$$

$$BW_{k+1} = \frac{BW_k}{2} \quad 1 \le k < n$$

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.

Wavelet Synthesis (Obsolete)

$$M_{i,k+1} = \frac{M_{i,k}}{2} \qquad 1 \le k < n$$

$$M_{i,n+1} = M_{i,n}$$

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.
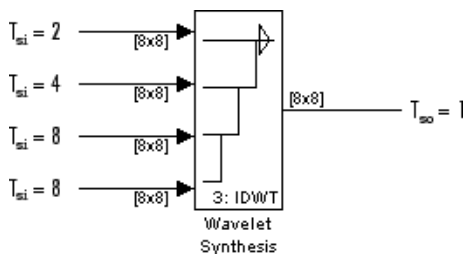
Frame-Based Operation

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.
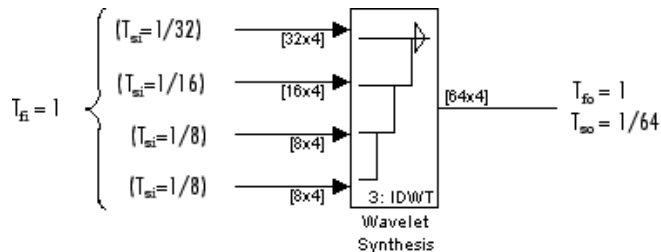
Zero Latency

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).

Wavelet Synthesis (Obsolete)

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" and the topic on models with multiple sample rates in the Simulink Coder documentation.
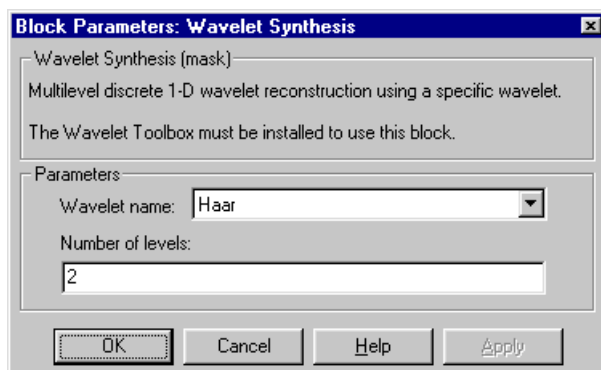
Dialog Box

The parameters displayed in the dialog box vary for different wavelet types. Only some of the parameters listed below are visible in the dialog box at any one time.

Wavelet name
> The wavelet used in the synthesis.

Wavelet order
> The order for the **Daubechies**, **Symlets**, and **Coiflets** wavelets. This parameter is available only when one of these wavelets is selected in the **Wavelet name** menu.

Filter order [synthesis / analysis]
> The filter orders for the synthesis and analysis stages of the **Biorthogonal** and **Reverse Biorthogonal** wavelets. For example, [2 / 6] selects a second-order synthesis stage and a sixth-order analysis stage. The **Filter order** parameter is available only when one of the above wavelets is selected in the **Wavelet name** menu.

Number of levels

The number of filter bank levels. An *n*-level structure has *n*+1 outputs.

References   Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types   • Double-precision floating point

See Also

Dyadic Synthesis Filter Bank	DSP System Toolbox
Wavelet Analysis (Obsolete)	DSP System Toolbox
wfilters	Wavelet Toolbox

Window Function

Purpose   Compute and/or apply window to input signal

Library   Signal Operations

dspsigops
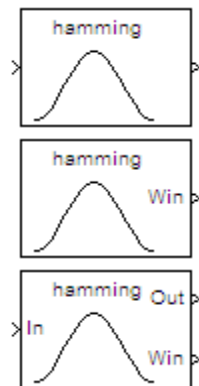
Description   The Window Function block computes a window and/or applies a
window to an input signal. The input signal can be a matrix or an
N-D array.

Operation Modes

The Window Function block has three modes of operation that you can
select via the **Operation** parameter. In each mode, the block first
creates a window vector w by sampling the window specified in the
Window type parameter at M discrete points. The operation modes
are:

- Apply window to input

  In this mode, the block computes an M-by-1 window vector w and
  applies it to the input. The output y always has the same dimension
  as the input.

  When the input is an M-by-N matrix u, the window is multiplied
  element-wise with each of the N channels in the input matrix u. This
  is equivalent to the following MATLAB code:

  ```
  y = repmat(w,1,N) .* u          % Equivalent MATLAB code
  ```

  The window is always applied to the first dimension:

  $$y(i, j, ..., k) = w(i) * u(i, j, ..., k) \qquad i = 1, ..., M, \;\; j = 1, ..., N, \;\; ..., \;\; k = 1, ..., P$$

  A length-M unoriented vector input is treated as an M-by-1 matrix.

- Generate window

In this mode, the block generates an unoriented window vector w with length M specified by the **Window length** parameter. The In port is disabled for this mode.

- `Generate and apply window`

  In this mode, the block generates an M-by-1 window vector w and applies it to the input. 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.

  - At the Win port, the block produces the M-by-1 window vector w.

  When the input is an M-by-N matrix u, the window is multiplied element-wise with each of the N channels in the input matrix u. This is equivalent to the following MATLAB code:

  ```
  y = repmat(w,1,N) .* u          % Equivalent MATLAB code
  ```

  The window is always applied to the first dimension:

  $$y(i, j,...,k) = w(i) * u(i, j,...,k) \qquad i = 1,...,M, \;\; j = 1,...,N, \;\; ..., \;\; k = 1,...,P$$

  A length-M 1-D vector input is treated as an M-by-1 matrix.

Window Type

The following table lists the available window types. For complete information about the window functions, consult the Signal Processing Toolbox documentation.

Window Function

Window Type	Description
Bartlett	Computes a Bartlett window. `w = bartlett(M)`
Blackman	Computes a Blackman window. `w = blackman(M)`
Boxcar	Computes a rectangular window. `w = rectwin(M)`
Chebyshev	Computes a Chebyshev window with stopband ripple `R`. `w = chebwin(M,R)`
Hamming	Computes a Hamming window. `w = hamming(M)`
Hann	Computes a Hann window (also known as a Hanning window). `w = hann(M)`
Hanning	Obsolete. This window type is included only for compatibility with older models. Use the `Hann` **Window type** instead of `Hanning` whenever possible.
Kaiser	Computes a Kaiser window with the Kaiser parameter `beta`. `w = kaiser(M,beta)`

Window Type	Description
Taylor	Computes a Taylor window. `w = taylorwin(M)`
Triang	Computes a triangular window. `w = triang(M)`

User Defined	Computes the user-defined window function specified by the entry in the **Window function name** parameter, usrwin.
	
	`w = usrwin(M) % Window takes no extra parameters`
	`w = usrwin(M,x_1,...,x_n) % Window takes extra`
	`parameters {x_1 ... x_n}`

Window Sampling

For the generalized-cosine windows (`Blackman`, `Hamming`, `Hann`, and `Hanning`), the **Sampling** parameter determines whether the window samples are computed in a periodic or a symmetric manner. For example, when **Sampling** is set to `Symmetric`, a Hamming window of length M is computed as

```
w = hamming(M)      % Symmetric (aperiodic) window
```

When **Sampling** is set to `Periodic`, the same window is computed as

```
w = hamming(M+1)      % Periodic (asymmetric) window
w = w(1:M)
```

Fixed-Point Data Types

The following diagram shows the data types used within the Window Function block for fixed-point signals for each of the three operating modes.

Apply window to input
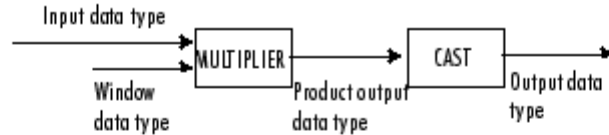
The input data type comes from the driving block. You can set the window, product output, and output data types in the block dialog. In this mode, the window vector is not output from the block.

Generate window

In this mode, the block acts as a source. The window vector is output in the window data type you specify in the block dialog.

Generate and apply window
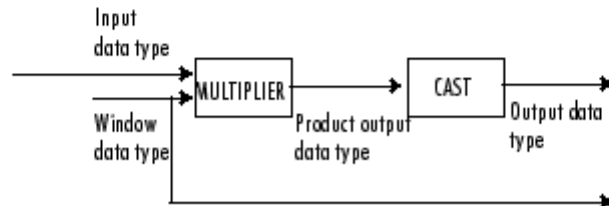
The input data type comes from the driving block. You can set the window, product output, and output data types in the block dialog. In this mode, the window vector is output from the block.

You can set the window, product output, and output data types in the block dialog box. For more information see the "Dialog Box" on page 2-1803 section.

Window Function

Examples

The following model uses the Window Function block to generate and apply a Hamming window to a 3-dimensional input array.

In this example, set the **Operation** mode of the Window Function block to `Generate and apply window`, so the block provides two outputs: the window vector w at the Win port, and the result of the multiplication y at the Out port.

Open the model by typing `ex_windowfunction_ref` at the MATLAB command line, and run it.
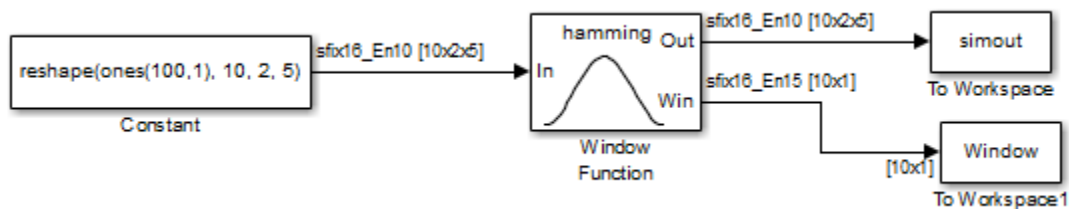
- The length of the first dimension of the input array is 10, so the Window Function block generates and outputs a Hamming window vector of length 10. To see the window vector generated by the Window Function block, type `w` at the MATLAB command line.

- To see the result of the multiplication, type `y` at the MATLAB command line.

Dialog Box

The **Main** pane of the Window Function block dialog appears as follows.
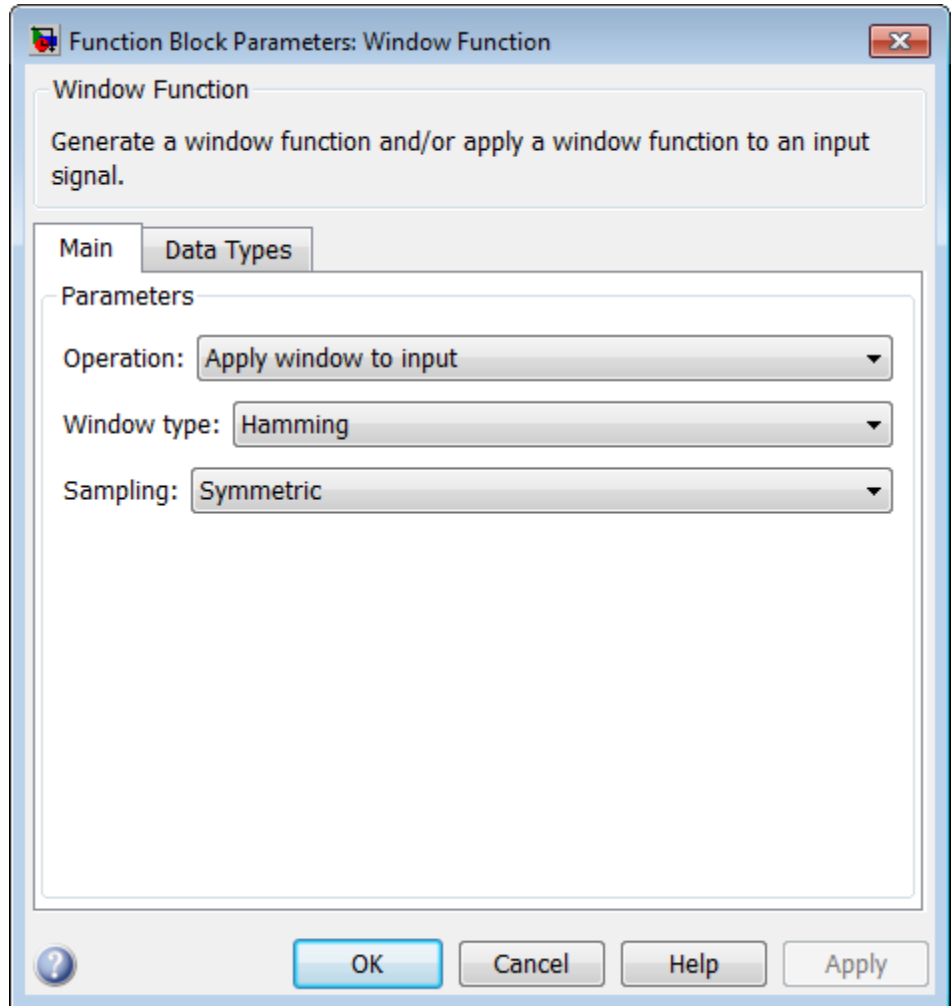
Window Function

Operation

Specify the block's operation, as discussed in "Operation Modes" on page 2-1796. The port configuration of the block is updated to match the setting of this parameter.

Window type

Specify the window type to apply, as listed in "Window Type" on page 2-1797. Tunable in simulation only.

Sampling

Specify the window sampling for generalized-cosine windows. This parameter is only visible when you select `Blackman`, `Hamming`, `Hann`, or `Hanning` for the **Window type** parameter. Tunable in simulation only.

Sample Mode

Specify the sample mode for the block, `Continuous` or `Discrete`, when it is in `Generate Window` mode. In the `Apply window to output` and `Generate and apply window` modes, the block inherits the sample time from its driving block. Therefore, this parameter is only visible when you select `Generate window` for the **Operation** parameter.

Sample time

Specify the sample time for the block when it is in `Generate window` and `Discrete` modes. In `Apply window to output` and `Generate and apply window` modes, the block inherits the sample time from its driving block. This parameter is only visible when you select `Discrete` for the **Sample Mode** parameter.

Window length

Specify the length of the window to apply. This parameter is only visible when you select `Generate window` for the **Operation** parameter. Otherwise, the window vector length is computed to match the length of the first dimension of the input.

Stopband attenuation in dB

Specify the level of stopband attenuation, R_s, in decibels. This parameter is only visible when you select `Chebyshev` for the **Window type** parameter. Tunable in simulation only.

Beta

Specify the `Kaiser` window β parameter. Increasing β widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. This parameter is only visible when you select `Kaiser` for the **Window type** parameter. Tunable in simulation only.

Number of sidelobes

Specify the number of sidelobes as a scalar integer value greater than zero. This parameter is only visible when you select `Taylor` for the **Window type** parameter.

Maximum sidelobe level relative to mainlobe (dB)

Specify, in decibels, the maximum sidelobe level relative to the mainlobe. This parameter must be a scalar less than or equal to zero. The default value of -30 produces sidelobes with peaks 30 dB down from the mainlobe peak. This parameter is only visible when you select `Taylor` for the **Window type** parameter.

Window function name

Specify the name of the user-defined window function to be calculated by the block. This parameter is only visible when you select `User defined` for the **Window type** parameter.

Specify additional arguments to the hamming function

Select to enable the **Cell array of additional arguments** parameter, when the user-defined window requires parameters other than the window length. This parameter is only visible when you select `User defined` for the **Window type** parameter.

Cell array of additional arguments

Specify the extra parameters required by the user-defined window function, besides the window length. This parameter is only available when you select the **Specify additional arguments to the hamming function** parameter. The entry must be a cell array.

The **Data Types** pane of the Window Function block dialog is discussed in the following sections:

Window Function

"Parameters for Generate Window Only Mode" on page 2-1806

"Parameters for Apply Window Modes" on page 2-1808

Parameters for Generate Window Only Mode

The **Data Types** pane of the Window Function block dialog appears as follows when the **Operation** parameter is set to Generate window.
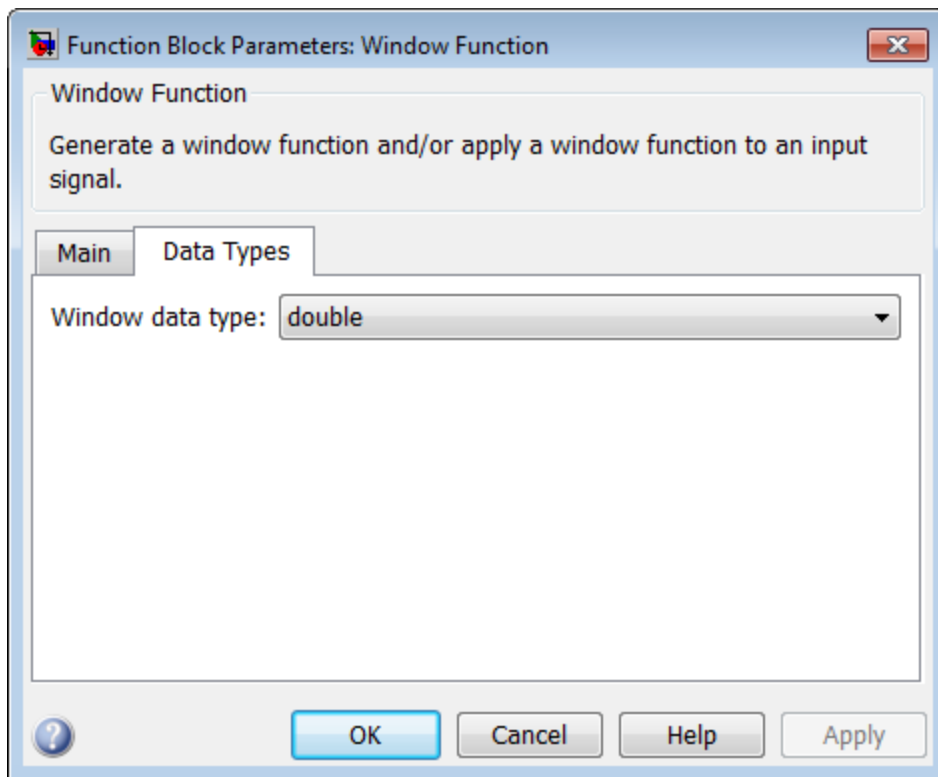
Window data type

Specify the window data type in one of the following ways:

- Choose double or single from the list.

- Choose `Fixed-point` to specify the window data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `User-defined` to specify the window data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `Inherit via back propagation` to set the window data type and scaling to match the following block.

Signed

Select to output a signed fixed-point signal. Otherwise, the signal is unsigned.

Word length

Specify the word length, in bits, of the fixed-point window data type. This parameter is only visible when you select `Fixed-point` for the **Window data type** parameter.

User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the Simulink Fixed Point functions `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac`. This parameter is only visible when you select `User-defined` for the **Window data type** parameter.

Set fraction length in output to

Specify the scaling of the fixed-point window data type by either of the following two methods:

- Choose `Best precision` to have the window data type scaling automatically set such that the output signal has the best possible precision.

- Choose `User-defined` to specify the window data type scaling in the **Fraction length** parameter.

This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Window data type** parameter, and when the specified window data type is a fixed-point data type.

Window Function

Fraction length

Specify the fraction length, in bits, of the fixed-point window data type. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Window data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Parameters for Apply Window Modes

The **Data Types** pane of the Window Function block dialog appears as follows when the **Operation** parameter is set to either `Apply window to input` or `Generate and apply window`.
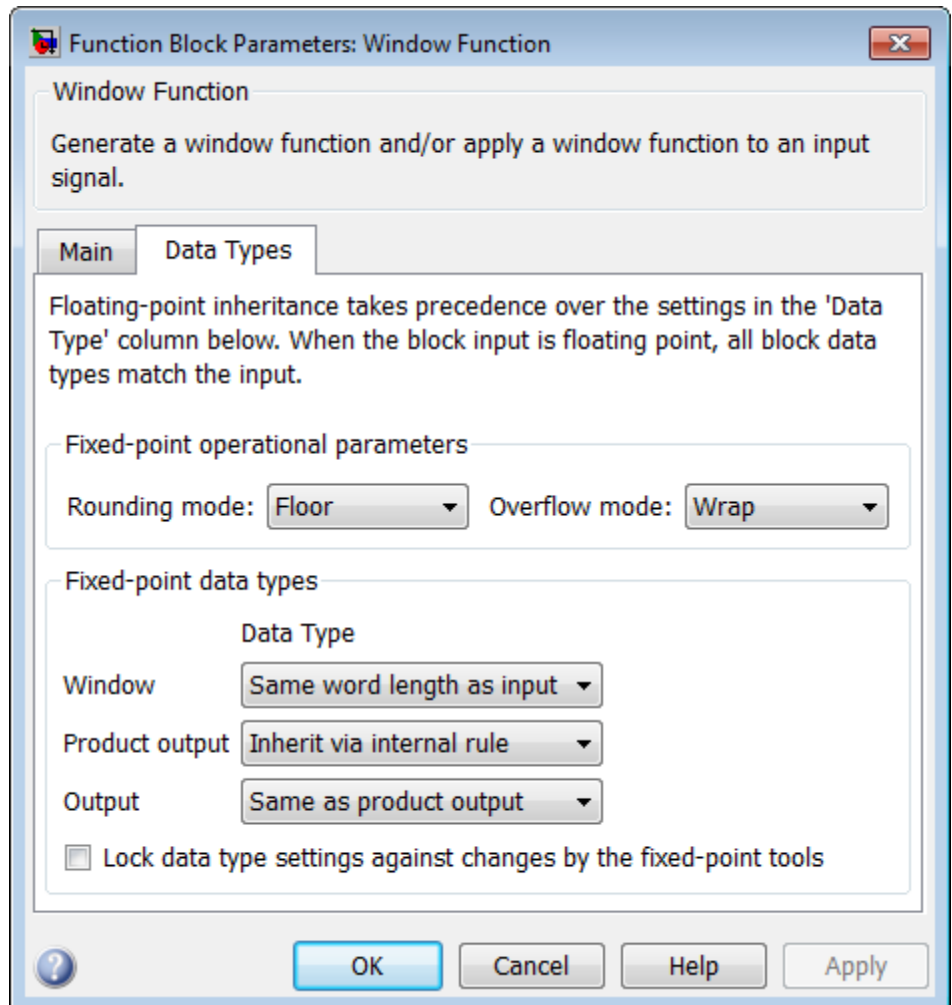
Rounding mode
Select the rounding mode for fixed-point operations.

The window vector *w* does not obey this parameter; it always rounds to `Nearest`.

Note The **Rounding mode** and **Overflow mode** settings have no effect on numerical results when both of the following conditions exist:

- **Product output data type** is `Inherit via internal rule`

- **Output data type** is `Same as product output`

With these data type settings, the block is effectively operating in full precision mode.

Overflow mode

Select the overflow mode for fixed-point operations.

The window vector *w* does not obey this parameter; it is always saturated.

Window

Choose how you specify the word length and fraction length of the window vector *w*.

When you select `Same word length as input`, the word length of the window vector elements is the same as the word length of the input. The fraction length is automatically set to the best precision possible.

When you select `Specify word length`, you can enter the word length of the window vector elements in bits. The fraction length is automatically set to the best precision possible.

When you select `Binary point scaling`, you can enter the word length and the fraction length of the window vector elements in bits.

When you select `Slope and bias scaling`, you can enter the word length, in bits, and the slope of the window vector elements. This block requires power-of-two slope and a bias of zero.

The window vector does not obey the **Rounding mode** and **Overflow mode** parameters; it is always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you want to designate the product output word and fraction lengths. See "Fixed-Point Data Types" on page 2-1690 for illustrations depicting the use of the product output data type in this block:

- When you select `Inherit via internal rule`, the product output word length and fraction length are calculated automatically. For information about how the product output word and fraction lengths are calculated when an internal rule is used, see "Inherit via Internal Rule".

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

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

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

Output

Choose how you specify the word length and fraction length of the output of the block:

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

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

Window Function

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

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

Lock data type settings against changes by the fixed-point tools
Select this parameter to prevent the fixed-point tools from overriding the data types you specify on the block mask.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Win	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit integers

See Also

FFT	DSP System Toolbox
bartlett	Signal Processing Toolbox
blackman	Signal Processing Toolbox
rectwin	Signal Processing Toolbox

chebwin	Signal Processing Toolbox
hamming	Signal Processing Toolbox
hann	Signal Processing Toolbox
kaiser	Signal Processing Toolbox
taylorwin	Signal Processing Toolbox
triang	Signal Processing Toolbox

Yule-Walker AR Estimator

Purpose   Compute estimate of autoregressive (AR) model parameters using Yule-Walker method

Library   Estimation / Parametric Estimation

dspparest3

Description
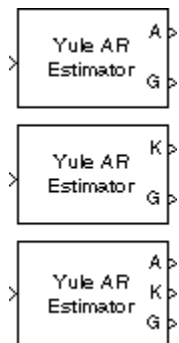
The Yule-Walker AR Estimator block uses the Yule-Walker AR method, also called the autocorrelation method, to fit an autoregressive (AR) model to the windowed input data by minimizing the forward prediction error in the least squares sense. This formulation leads to the Yule-Walker equations, which are solved by the Levinson-Durbin recursion. Block outputs are always nonsingular.

The Yule-Walker AR Estimator block can output the AR model coefficients as polynomial coefficients, reflection coefficients, or both. Each channel of 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 signal that is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input frame.

$$H(z) = \frac{\sqrt{G}}{A(z)} = \frac{\sqrt{G}}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

When you select **Inherit estimation order from input dimensions**, the order p of the all-pole model is one less than the length of each input channel. Otherwise, the order is the value specified by the **Estimation order** parameter. To guarantee a valid output, you must set the **Estimation order** parameter to be a scalar less than or equal to half the input channel length. The Yule-Walker AR Estimator and Burg AR Estimator blocks return similar results for large frame sizes.

When **Output(s)** is set to A, port A is enabled. For each channel, port A outputs a column of length $p+1$ that contains the normalized estimate of the AR model coefficients in descending powers of z

```
[1 a(2) ... a(p+1)]
```

When **Output(s)** is set to K, port K is enabled. For each channel, port K outputs a length-p column whose elements are the AR model reflection coefficients. When **Output(s)** is set to A and K, both port A and K are enabled, and each port outputs the respective AR model coefficients for each channel.

The square of the model gain, G, is provided at port G. G is a scalar for each channel.

See the Burg AR Estimator block reference page for a comparison of the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

Dialog Box

Block Parameters: Yule-Walker AR Estimator

Yule-Walker AR Estimator (mask) (link)

Frame-based parametric AR estimation using the Yule-Walker method. The AR model coefficients are given in A (polynomial coefficients) and/or K (reflection coefficients). The square of the model gain is given in G.

Parameters

Output(s): A

☑ Inherit estimation order from input dimensions

Estimation order:

4

OK   Cancel   Help   Apply
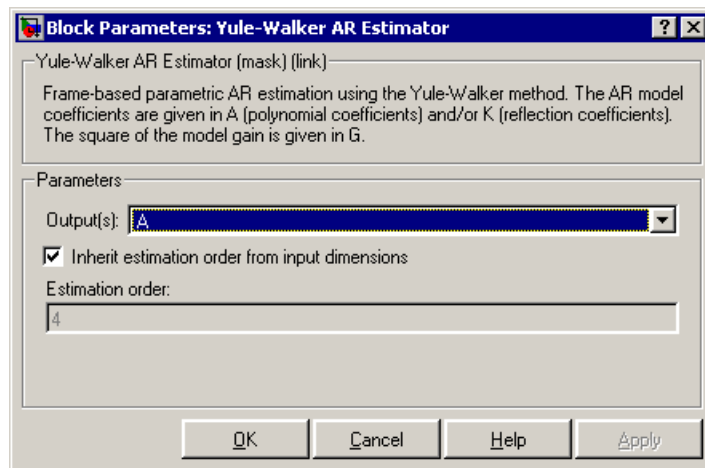
Output(s)
> The type of AR model coefficients output by the block. The block can output polynomial coefficients (A), reflection coefficients (K), or both (A and K).

Inherit estimation order from input dimensions
> When selected, sets the estimation order p to one less than the length of each input channel.

Yule-Walker AR Estimator

Estimation order
> The order of the AR model, p. This parameter is enabled when you do not select **Inherit estimation order from input dimensions**.

References   Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
A	• Double-precision floating point
	• Single-precision floating point
K	• Double-precision floating point
	• Single-precision floating point
G	• Double-precision floating point
	• Single-precision floating point

See Also

Burg AR Estimator	DSP System Toolbox
Covariance AR Estimator	DSP System Toolbox
Modified Covariance AR Estimator	DSP System Toolbox
Yule-Walker Method	DSP System Toolbox
aryule	Signal Processing Toolbox

Purpose   Design and apply IIR filter

Library   dspobslib

Description
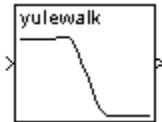
Note The Yule-Walker IIR Filter Design block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Yule-Walker IIR Filter Design block designs a recursive (ARMA) digital filter with arbitrary multiband magnitude response, and applies it to a discrete-time input using the Direct-Form II Transpose Filter block. The filter design, which uses the Signal Processing Toolbox yulewalk function, performs a least-squares fit to the specified frequency response.

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

The **Band-edge frequency vector** parameter is a vector of frequency points in the range 0 to 1, where 1 corresponds to half the sample frequency. The first element of this vector must be 0 and the last element 1, and intermediate points must appear in ascending order. The **Magnitudes at these frequencies** parameter is a vector containing the desired magnitude response at the points specified in the **Band-edge frequency vector**.

Note that, unlike the Remez FIR Filter Design block, each frequency-magnitude pair specifies the junction of two adjacent frequency bands, so there are no "don't care" regions.

Yule-Walker IIR Filter Design (Obsolete)

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:
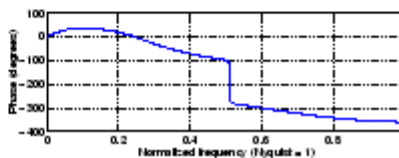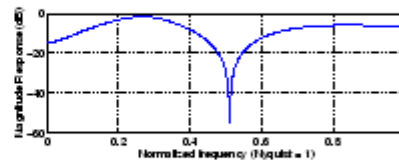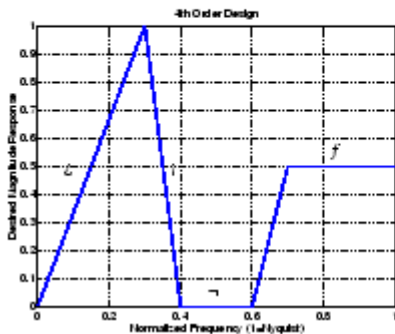
When specifying the **Band-edge frequency vector** and **Magnitudes at these frequencies** vectors, avoid excessively sharp transitions from passband to stopband. You may need to experiment with the slope of the transition region to get the best filter design.

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**
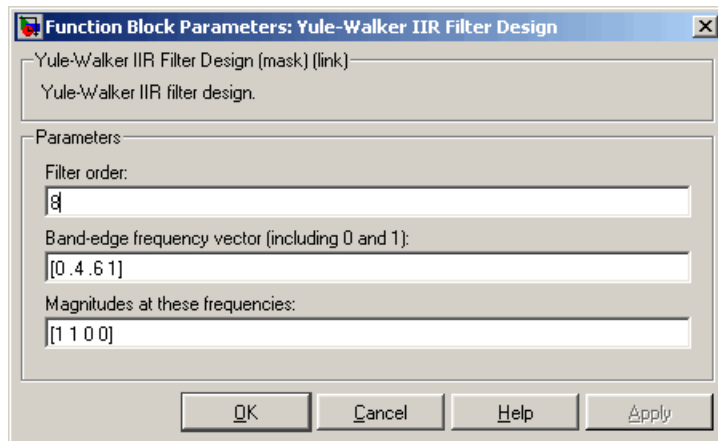
Filter order

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.

Yule-Walker Method

Purpose

Power spectral density estimate using Yule-Walker method

Library

Estimation / Power Spectrum Estimation

dspspect3

Description
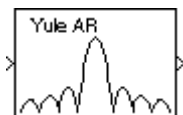
The Yule-Walker Method block estimates the power spectral density (PSD) of the input using the Yule-Walker AR method. This method, also called the *autocorrelation method*, fits an autoregressive (AR) model to the windowed input data. It does so by minimizing the forward prediction error in the least squares sense. This formulation leads to the Yule-Walker equations, which the Levinson-Durbin recursion solves. Block outputs are always nonsingular.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only). This input represents a frame of consecutive time samples from a single-channel signal. The block outputs a column vector containing the estimate of the power spectral density of the signal at N_{fft} equally spaced frequency points. The frequency points are in the range $[0, F_s)$, where F_s is the sampling frequency of the signal.

When you select **Inherit estimation order from input dimensions**, the order of the all-pole model is one less that the input frame size. Otherwise, the **Estimation order** parameter value specifies the order. To guarantee a valid output, the **Estimation order** parameter must be less than or equal to half the input vector length. The block computes the spectrum from the FFT of the estimated AR model parameters.

Selecting the **Inherit FFT length from estimation order** parameter specifies that N_{fft} is one greater than the estimation order. Clearing the **Inherit FFT length from estimation order** check box allows you to use the **FFT length** parameter to specify N_{fft} as a power of 2. The block zero-pads or wraps the input to N_{fft} before computing the FFT. The output is always sample based.

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must 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 equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

See the Burg Method block reference for a comparison of the Burg Method, Covariance Method, Modified Covariance Method, and Yule-Walker AR Estimator blocks. The Yule-Walker AR Estimator and Burg Method blocks return similar results for large buffer lengths.

Yule-Walker Method
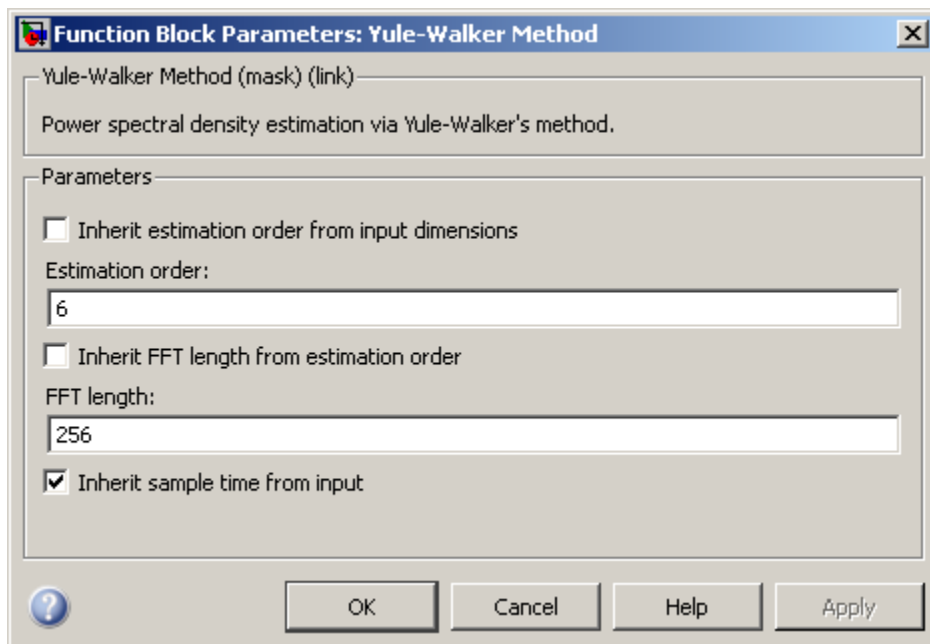
Dialog Box

Inherit estimation order from input dimensions
> When you select this option, it sets the estimation order to one less than the length of the input vector.

Estimation order
> Specify the order of the AR model. This parameter is only visible when you clear the **Inherit estimation order from input dimensions** check box.

Inherit FFT length from estimation order
> When you select the **Inherit FFT length from estimation order** check box, the FFT length is one greater than the estimation order. To specify the number of points on which to perform the FFT, clear the **Inherit FFT length from estimation order** check box. You can then specify a power-of-two FFT length using the **FFT length** parameter.

FFT length

Enter the number of data points on which to perform the FFT, N_{fft}. When N_{fft} is larger than the input frame size, the block zero-pads each frame as needed. When N_{fft} is smaller than the input frame size, the block wraps each frame as needed. This parameter becomes visible only when you clear the **Inherit FFT length from input dimensions** check box.

Inherit sample time from input

When you select the **Inherit sample time from input** check box, the block computes the frequency data from the sample period of the input signal. For the block to produce valid output, the following conditions must 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 equals the sample period of the original time series.

If these conditions do not hold, clear the **Inherit sample time from input** check box. You can then specify a sample time using the **Sample time of original time series** parameter.

Sample time of original time series

Specify the sample time of the original time-domain signal. This parameter becomes visible only when you clear the **Inherit sample time from input** check box.

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.

Orfanidis, S. J. *Introduction to Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 1995.

Yule-Walker Method

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type.

See Also

Burg Method	DSP System Toolbox
Covariance Method	DSP System Toolbox
Levinson-Durbin	DSP System Toolbox
Autocorrelation LPC	DSP System Toolbox
Short-Time FFT	DSP System Toolbox
Yule-Walker AR Estimator	DSP System Toolbox
spectrum.yulear	Signal Processing Toolbox

See "Spectral Analysis" for related information.

Purpose   Count number of times signal crosses zero in single time step

Library   Signal Operations

dspsigops
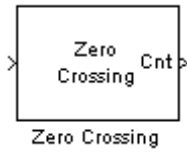
Description   The Zero Crossing block concludes that a signal in a given channel has passed through zero if it meets any of the following criteria, where x_i is the current signal value, x_{i-1} is the previous signal value, and so on:

```
Zero      Cnt
Crossing
```

Zero Crossing

- $x_i < 0$ and $x_{i-1} > 0$

- $x_i > 0$ and $x_{i-1} < 0$

- For some positive integer L, $x_i < 0$, $x_{i-l} = 0$, and $x_{i-L-1} > 0$, where $0 \leq l \leq L$.

- For some positive integer L, $x_i > 0$, $x_{i-l} = 0$, and $x_{i-L-1} < 0$, where $0 \leq l \leq L$.

For the first input value, x_{i-1} and x_{i-2} are zero. The block outputs the number of times the signal crosses zero in a single time step at the Cnt port.

The input to this block must be a real-valued fixed-point or floating-point signal. If you set the **Input processing** parameter to Elements as channels (sample based), the block treats each element of the input as a time-varying channel. If you set the **Input processing** parameter to Columns as channels (frame based), the block treats each column of the input as an independent channel.

Examples   The following example, ex_zero_crossing illustrates the behavior of the Zero Crossing block.

To run the model for one time step, the **Stop time** is set to 0.

1

Run the model.

Because the signal passes through zero once during the first time step, the Zero Crossing block finds one zero crossing. The number of detected zero crossings in the first time step is shown in the Display block in the following figure.
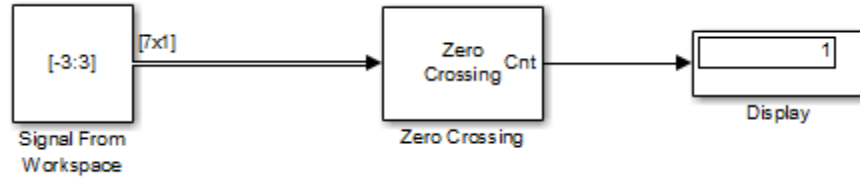
2

To run the model for two time steps, change the simulation **Stop time** to 1. To do so, open the Configuration Parameters dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu. In the **Solver** pane, set **Stop time** to 1.

3

Run the model.

The Zero Crossing block remembers that the last value of the last frame was 3. Therefore, the signal passes through zero twice during the second time step. It passes through zero while going from 3 to -3, and it passes through zero again while going from -3 to 3. The Zero Crossing block finds two zero crossings in the second time step as shown in the Display block in the following figure.
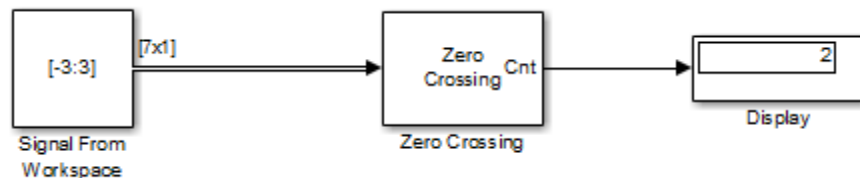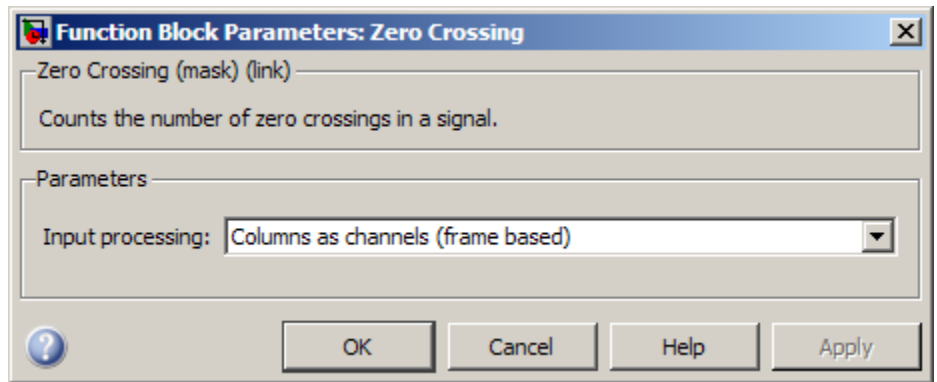
Dialog Box

Input processing

Specify how the block should process the input. You can set this parameter to one of the following options:

- Columns as channels (frame based) — When you select this option, the block treats each column of the input as a separate channel.

- Elements as channels (sample based) — When you select this option, the block treats each element of the input as a separate channel.

Note The Inherited (this choice will be removed - see release notes) option will be removed in a future release. See "Frame-Based Processing" in the *DSP System Toolbox Release Notes* for more information.

Zero Crossing

ERROR

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating-point
	• Single-precision floating-point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Cnt	• 32-bit unsigned integers

See Also

Hit Crossing   Simulink

2-1828

3

System object Reference

Define New System Objects

getDiscreteStateImpl	Discrete state property values
getNumInputsImpl	Number of input arguments passed to step and setup methods
getNumOutputsImpl	Number of outputs returned by method
isDoneImpl	End-of-data flag
isInactivePropertyImpl	Active or inactive flag for properties
loadObjectImpl	Load saved System object from MAT file
matlab.System	Base class for System objects
matlab.system.mixin.FiniteSource	Finite source mixin class
matlab.system.StringSet	Set of valid string values
processTunedPropertiesImpl	Action when tunable properties change
releaseImpl	Release resources
resetImpl	Reset System object™ states
saveObjectImpl	Save System object in MAT file
setProperties	Set property values from name-value pair inputs
setupImpl	Initialize System object
stepImpl	System output and state update equations
validateInputsImpl	Validate inputs to step method
validatePropertiesImpl	Validate property values

Estimation

dsp.BurgAREstimator	Estimate of autoregressive (AR) model parameters using Burg method
dsp.BurgSpectrumEstimator	Parametric spectral estimate using Burg method
dsp.CepstralToLPC	Convert cepstral coefficients to linear prediction coefficients
dsp.LevinsonSolver	Solve linear system of equations using Levinson-Durbin recursion
dsp.LPCToAutocorrelation	Convert linear prediction coefficients to autocorrelation coefficients
dsp.LPCToCepstral	Convert linear prediction coefficients to cepstral coefficients
dsp.LPCToLSF	Convert linear prediction coefficients to line spectral frequencies
dsp.LPCToLSP	Convert linear prediction coefficients to line spectral pairs
dsp.LPCToRC	Convert linear prediction coefficients to reflection coefficients
dsp.LSFToLPC	Convert line spectral frequencies to linear prediction coefficients
dsp.LSPToLPC	Convert line spectral pairs to linear prediction coefficients
dsp.RCToAutocorrelation	Convert reflection coefficients to autocorrelation coefficients
dsp.RCToLPC	Convert reflection coefficients to linear prediction coefficients

Filtering

dsp.AllpoleFilter	IIR Filter with no zeros
dsp.BiquadFilter	IIR filter using biquadratic structures
dsp.BlockLMSFilter	Output, error, and weights using Block LMS adaptive algorithm
dsp.CICDecimator	Decimate input using Cascaded Integrator-Comb filter
dsp.CICInterpolator	Interpolate signal using Cascaded Integrator-Comb filter
dsp.DigitalFilter	Static or time-varying digital filter
dsp.DyadicAnalysisFilterBank	Dyadic analysis filter bank
dsp.DyadicSynthesisFilterBank	Reconstruct signals from subbands
dsp.FIRDecimator	FIR polyphase decimator
dsp.FIRFilter	Static or time-varying FIR filter
dsp.FIRInterpolator	Polyphase FIR interpolator
dsp.FIRRateConverter	Sample rate converter
dsp.IIRFilter	Infinite Impulse Response (IIR) filter
dsp.LMSFilter	LMS adaptive filter
dsp.SubbandAnalysisFilter	Decompose signal into high-frequency and low-frequency subbands
dsp.SubbandSynthesisFilter	Reconstruct signal from high-frequency and low-frequency subbands

Math Functions

dsp.ArrayVectorAdder	Add array to vector along specified dimension
dsp.ArrayVectorDivider	Divide array by vector along specified dimension
dsp.ArrayVectorMultiplier	Multiply array by vector along specified dimension
dsp.ArrayVectorSubtractor	Subtract vector from array along specified dimension
dsp.CumulativeProduct	Cumulative product of channel, column, or row elements
dsp.CumulativeSum	Cumulative sum of channel, column, or row elements
dsp.LDLFactor	Factor square Hermitian positive definite matrices into components
dsp.LevinsonSolver	Solve linear system of equations using Levinson-Durbin recursion
dsp.LowerTriangularSolver	Solve lower-triangular matrix equation
dsp.LUFactor	Factor square matrix into lower and upper triangular matrices
dsp.Normalizer	Vector normalization along specified dimension
dsp.UpperTriangularSolver	Solve upper-triangular matrix equation

Quantizers

dsp.ScalarQuantizerDecoder	Convert each index value into quantized output value
dsp.ScalarQuantizerEncoder	Associate input value with index value of quantization region
dsp.UniformDecoder	Decode integer input into floating-point output
dsp.UniformEncoder	Quantize and encode floating-point input into integer output
dsp.VectorQuantizerDecoder	Vector quantizer codeword for given index value
dsp.VectorQuantizerEncoder	Vector quantization encoding

Signal Management

dsp.Buffer	Buffer input signal
dsp.Counter	Count up or down through specified range of numbers
dsp.DelayLine	Rebuffer sequence of inputs with one-sample shift

Signal Operations

dsp.Convolver	Convolution of two inputs
dsp.Delay	Delay input by specified number of samples or frames
dsp.DigitalDownConverter	Digitally downconvert input signal
dsp.DigitalUpConverter	Digitally upconvert input signal
dsp.Interpolator	Linear or FIR interpolation
dsp.NCO	Generate real or complex sinusoidal signals
dsp.PeakFinder	Determine extrema (maxima or minima) in input signal
dsp.PhaseUnwrapper	Unwrap signal phase
dsp.VariableFractionalDelay	Delay input by time-varying fractional number of sample periods
dsp.VariableIntegerDelay	Delay input by time-varying integer number of sample periods
dsp.Window	Window object
dsp.ZeroCrossingDetector	Zero crossing detector

Sinks

dsp.AudioFileWriter	Write audio samples to audio file
dsp.AudioPlayer	Play audio data using computer's audio device
dsp.MatFileWriter	Write MAT file
dsp.SignalSink	Log simulation data in buffer
dsp.SpectrumAnalyzer	Display frequency spectrum of time-domain signals
dsp.TimeScope	Time domain signal display
dsp.UDPSender	Send UDP packets to network
signalblks.MultimediaFileWriter	Write video frames, audio samples, or both to multimedia file

Sources

dsp.AudioFileReader	Read audio samples from audio file
dsp.AudioRecorder	Record audio data using computer's audio device
dsp.Chirp	Generate swept-frequency cosine (chirp) signal
dsp.MatFileReader	Read MAT file
dsp.SignalSource	Import variable from workspace
dsp.SineWave	Discrete-time sinusoid
dsp.UDPReceiver	Receive UDP packets from network
signalblks.MultimediaFileReader	Read video frames, audio samples, or both from multimedia file

Measurements and Statistics

dsp.Autocorrelator	Autocorrelation sequence
dsp.Crosscorrelator	Cross-correlation of two inputs
dsp.Histogram	Histogram of input or sequence of inputs
dsp.Maximum	Find maximum value of input or sequence of inputs
dsp.Mean	Find mean value of input or sequence of inputs
dsp.Median	Median value of input
dsp.Minimum	Find minimum values of input or sequence of inputs
dsp.PeakToPeak	Peak-to-peak value
dsp.PeakToRMS	Peak-to-root-mean-square value of vector
dsp.PulseMetrics	Pulse metrics of bilevel waveforms
dsp.RMS	Root mean square of vector elements
dsp.StandardDeviation	Standard deviation of input or sequence of inputs
dsp.StateLevels	State-level estimation for bilevel rectangular waveform
dsp.TransitionMetrics	Transition metrics of bilevel waveforms
dsp.Variance	Variance of input or sequence of inputs

Transforms

dsp.AnalyticSignal	Analytic signals of discrete-time inputs
dsp.DCT	Discrete cosine transform (DCT)
dsp.FFT	Discrete Fourier transform
dsp.IDCT	Inverse discrete cosine transform (IDCT)
dsp.IFFT	Inverse discrete Fourier transform (IDFFT)

Analysis Methods for Filter System Objects

Analysis Methods for Filter System Objects

Method	Description
Single-rate and Multirate Analysis	
fvtool	Filter visualization tool
info	Filter information
freqz	Frequency response of a discrete-time filter
phasez	Phase response of a discrete-time filter
zerophase	Zero-phase response of a discrete-time filter
grpdelay	Group delay of a discrete-time filter
phasedelay	Phase delay of a discrete-time filter
impz	Impulse response of a discrete-time filter
stepz	Step response of a discrete-time filter
zplane	Pole/Zero plot
cost	Cost estimate
measure	measure filter response
order	Filter order
firtype	Determine the type (1–4) of a linear phase FIR filter
coeffs	Return filter coefficients
Multirate Analysis	
polyphase	Polyphase decomposition of multirate filters
gain	Gain of a Cascaded Integrator-Comb (CIC) filter

Method	Description
Second-order Sections	
`scale`	Scale second-order sections
`scalecheck`	Check scale of second-order sections
`scaleopts`	Create options object for sos scaling
`cumsec`	Vector of cumulative second-order section filters
`reorder`	Reorder second-order sections
`sos`	Convert IIR filter to Biquad filter
Code Generation	
`realizemdl`	Filter realization diagram
Fixed-point (supported by FIRFilter, IIRFilter, AllpoleFilter, and BiquadFilter only)	
`noisepsd`	Power spectral density of filter output due to roundoff noise
`noisepsdopts`	Create options object for noisepsd computation
`freqrespest`	Frequency response estimate via filtering
`freqrespopts`	Create options object for frequency response estimate
Other	
`isallpass`	True for allpass discrete-time filter
`islinphase`	True for linear discrete-time filter
`ismaxphase`	True if maximum phase
`isminphase`	True if minimum phase
`isreal`	True for discrete-time filter with real coefficients
`isstable`	True if the filter is stable

Method	Description
isfir	True if the filter is FIR
issos	True if filter is in second order sections form
specifyall	Specify all fixed-point properties

Alphabetical List

dsp.AllpoleFilter

Purpose　　　IIR Filter with no zeros

Description　　The `AllpoleFilter` object filters each channel of the input using Allpole filter implementations.

To filter each channel of the input:

1 Define and set up your Allpole filter. See "Construction" on page 5-2.

2 Call `step` to filter each channel of the input according to the properties of `dsp.AllpoleFilter`. The behavior of `step` is specific to each object in the toolbox.

Construction　　`HAllpole = dsp.AllpoleFilter` returns an Allpole filter System object, `HAllpole`, which independently filters each channel of the input over successive calls to the `step` method. This System object uses a specified Allpole filter implementation.

`HAllpole = dsp.AllpoleFilter('`*PropertyName*`',`*PropertyValue*`, ...)` returns an Allpole filter System object, `HAllpole`, with each property set to the specified value.

Properties　　**Structure**

Filter structure

Specify the filter structure.

You can specify the filter structure as one of | `Direct form` | `Direct form transposed` | `Lattice AR`. The default is `Direct form`. Analysis methods are not supported for fixed-point processing if the structure is `Direct form` or `Direct form transposed`. This property is nontunable.

Denominator

Filter denominator coefficients

Specify the denominator coefficients as a real or complex numeric row vector. This property is applicable when the Structure

property is set to one of `Direct form` | `Direct form transposed`. The default value of this property is `[1 0.1]`. This property is tunable.

ReflectionCoefficients

Lattice filter coefficients

Specify the lattice filter coefficients as a real or complex numeric row vector. This property is applicable when the Structure property is set to `Lattice AR`. The default value of this property is `[0.2 0.4]`. This property is tunable.

InitialConditions

Initial conditions for the filter states

Specify the initial conditions of the filter states. The default value is `0`.

You can specify the initial conditions as a scalar, vector, or matrix. If you specify a scalar value, this System object initializes all delay elements in the filter to that value. You can also specify a vector whose length equals the number of delay elements in the filter. When you do so, each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector of initial conditions to each channel of the input signal.

You can also specify a matrix with the same number of rows as the number of delay elements in the filter and one column for each channel of the input signal. In this case, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. This property is tunable.

FrameBasedProcessing

Process input as frames or as samples

Set this property to `true` to enable frame-based processing. When this property is `true`, the Allpole filter treats each column as an independent channel. Set this property to `false` to enable

dsp.AllpoleFilter

sample-based processing. When this property is `false`, the Allpole filter treats each element of the input as an individual channel. The default is `true`. This property is nontunable.

CoefficientsDataType

Denominator coefficients word- and fraction-length designations

Specify the denominator coefficients fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property is nontunable.

ReflectionCoefficientsDataType

Reflection coefficients word- and fraction-length designations

Specify the reflection coefficients fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property is nontunable.

Fixed-Point Properties

ProductDataType

Product word- and fraction-length designations

Specify the product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`. This property is nontunable.

AccumulatorDataType

Accumulator word- and fraction-length designations

Specify the accumulator fixed-point data type to one of | `Full precision` | `Same as input` | `Same as product` | `Custom` |. The default is `Full precision`. This property is nontunable.

OutputDataType

Output word- and fraction-length designations

Specify the output fixed-point data type as one of | `Same as accumulator` | `Same as input` | `Custom` |. The default is `Same as input`. This property is nontunable.

StateDataType

State word- and fraction-length designations

Specify the state fixed-point data type as one of | `Same as input` | `Same as accumulator` | `Custom`. The default is `Same as accumulator`. This property is nontunable.

CustomCoefficientsDataType

Custom denominator word- and fraction-lengths

Specify the denominator coefficients fixed-point type as an autosigned numerictype object. This property is applicable when the `CoefficientsDataType` property is `Custom`. The default value of this property is numerictype ([],16,15). This property is nontunable.

CustomReflectionCoefficientsDataType

Custom reflection coefficients word- and fraction-lengths

Specify the denominator coefficients fixed-point type as an autosigned numerictype object. This property is applicable when the `ReflectionCoefficientsDataType` property is `Custom`. The default value of this property is numerictype ([],16,15). This property is nontunable.

CustomProductDataType

Custom Product word- and fraction-lengths

Specify the product fixed-point type as an autosigned scaled numerictype object. This property applies when you set the `ProductDataType` property to `Custom`. The default is numerictype ([],32,30). This property is nontunable.

CustomAccumulatorDataType

Custom accumulator word- and fraction-lengths

Specify the accumulator fixed-point type as an autosigned scaled numerictype object. This property applies when you set the AccumulatorDataType property to Custom. The default is numerictype([],32,30). This property is nontunable.

CustomStateDataType

Custom state word- and fraction-lengths

Specify the state fixed-point type as an autosigned scaled numerictype object. This property applies when you set the StateDataType property to Custom. The default is numerictype([],16,15). This property is nontunable.

CustomOutputDataType

Custom output word- and fraction-lengths

Specify the output fixed-point type as an autosigned scaled numerictype object. This property applies when you set the OutputDataType property to Custom. The default is numerictype([],16,15). This property is nontunable.

Methods		
	clone	Create Allpole filter with same property values
	freqz	Frequency response
	fvtool	Open filter visualization tool
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	impz	Impulse response
	isLocked	Locked status for input attributes and nontunable properties
	phasez	Unwrapped phase response

release	Allow property value and input characteristics changes
reset	Reset internal states of Allpole filter
step	Filter input with Allpole filter object

More "Analysis Methods for Filter System Objects" on page 4-2.

Note In AllpoleFilter, analysis methods are not supported for fixed-point processing if the structure is Direct form or Direct form transposed.

Examples   Use an IIR filter to apply a lowpass filter to a waveform with two sinusoidal frequencies.

```
t = [0:63]./32e3;
xin = (sin(2*pi*4e3*t)+sin(2*pi*12e3*t)) / 2;

hSR = dsp.SignalSource(xin', 4);
hLog = dsp.SignalSink;
hallpole = dsp.AllpoleFilter;
a = lpc(xin,6);
hallpole.Denominator = a;

while ~isDone(hSR)
     input = step(hSR);
     filteredOutput = step(hallpole,input);
     step(hLog,filteredOutput);
end

filteredResult = hLog.Buffer;
sper = spectrum.periodogram;
psd(sper,filteredResult,'Fs',32e3)
```

dsp.AllpoleFilter

Algorithms

This object implements the algorithm, inputs, and outputs described on the Allpole Filter block reference page. The object properties correspond to the block parameters.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the `FrameBasedProcessing` property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also

`dsp.BiquadFilter` | `dsp.FIRFilter` | `dsp.IIRFilter`

Purpose   Create Allpole filter with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a Allpole filter object, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

dsp.AllpoleFilter.freqz

Purpose   Frequency response

Syntax   ```
[h,w] = freqz(H)
[h,w] = freqz(H,n)
[h,w] = freqz(H,Name,Value)
freqz(H)
```

**Description**  `[h,w] = freqz(H)` returns the complex, 8192–element frequency
response vector `h`, and the corresponding frequencies `w` in
radians/sample.

`[h,w] = freqz(H,n)` returns the complex, n-element frequency
response vector `h`, and the corresponding frequencies `w` in
radians/sample, using `n` samples.

`[h,w] = freqz(H,Name,Value)` returns the frequency response and
the corresponding frequencies, with additional options specified by one
or more Name, Value pair arguments.

`freqz(H)` uses FVTool to plot the magnitude and unwrapped phase of
the frequency response of the filter System object `H`.

**Purpose**     Open filter visualization tool

**Syntax**      `fvtool(H)`
                `fvtool(H,'Arithmetic',ARITH,...)`

**Description**     `fvtool(H)` performs an analysis and computes the magnitude response of the filter System object `H`.

`fvtool(H,'Arithmetic',ARITH,...)` analyzes the filter System object `H`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

# dsp.AllpoleFilter.getNumInputs

**Purpose**      Number of expected inputs to step method

**Syntax**       N = getNumInputs(H)

**Description**  N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

**Purpose**        Number of outputs of step method

**Syntax**         N = getNumOutputs(H)

**Description**    N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

**Purpose**        Impulse response

**Syntax**
```
[h,t] = impz(H)
[h,t] = impz(H,Name,Value)
impz(H)
```

**Description**    `[h,t] = impz(H)` returns the impulse response h, and the corresponding time points t at which the impulse response of H is computed.

`[h,t] = impz(H,Name,Value)` returns the impulse response h, and the corresponding time points t , with additional options specified by one or more Name, Value pair arguments.

`impz(H)` uses FVTool to plot the impulse response of the filter System object H.

**Note**  You can use impz for both real and complex filters. When you omit the output arguments, impz plots only the real part of the impulse response.

**Purpose**     Locked status for input attributes and nontunable properties

**Syntax**      isLocked(H)

**Description**     isLocked(H) returns the locked state of the Allpole filter.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

**Purpose**    Unwrapped phase response

**Syntax**
```
[phi,w] = phasez(H)
[phi,w] = phasez(H,n)
[phi,w] = phasez(H,Name,Value)
phasez(H)
```

**Description**    `[phi,w] = phasez(H)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(H,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(H,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(H)` uses FVTool to plot the phase response of the filter System object `H`.

**Purpose**        Allow property value and input characteristics changes

**Syntax**         release(H)

**Description**    release(H) releases system resources, such as memory, file handles,
                   and hardware connections, and lets you change any properties or input
                   characteristics.

---

**Note**  You can use the release method on a System object in code
generated from MATLAB, but once you release its resources, you cannot
use that System object again.

---

# dsp.AllpoleFilter.reset

**Purpose**      Reset internal states of Allpole filter

**Syntax**      reset(H)

**Description**      reset(H) resets the filter states of the Allpole filter object, H, to their
initial values of 0. The initial filter state values correspond to the initial
conditions for the difference equation defining the filter. After the step
method applies the Allpole filter object to nonzero input data, the states
may be nonzero. Invoking the step method again without first invoking
the reset method may produce different outputs for an identical input.

For example:

```
n = 0:100;
x = cos(0.2*pi*n)+sin(0.8*pi*n);
H = dsp.AllpoleFilter;
a = lpc(x,20);
H.Denominator = a;
y = step(H,x);
% Filter states are nonzero
% Invoke step method again without resetting states
y1 = step(H,x);
isequal(y,y1) % returns 0
% Now reset filter states to 0
reset(H)
% Invoke step method
y2 = step(H,x);
isequal(y,y2) % returns a 1
```

Purpose   Filter input with Allpole filter object

Syntax   Y = step(H,X)

Description   Y = step(H,X) filters the real or complex input signal X using the
                 Allpole filter, H, to produce the output Y . When the input data is of a
                 fixed-point type, it must be signed. The Allpole filter object operates on
                 each channel of the input signal independently over successive calls
                 to step method.

                 Note The object performs an initialization the first time the step
                 method is executed. This initialization locks nontunable properties and
                 input specifications, such as dimensions, complexity, and data type
                 of the input data. If you change a nontunable property or an input
                 specification, the System object issues an error. To change nontunable
                 properties or inputs, you must first call the release method to unlock
                 the object.

dsp.AnalyticSignal

Purpose   Analytic signals of discrete-time inputs

Description   The `AnalyticSignal` object computes analytic signals of discrete-time inputs. The real part of the analytic signal in each channel is a replica of the real input in that channel, and the imaginary part is the Hilbert transform of the input. In the frequency domain, the analytic signal doubles the positive frequency content of the original signal while zeroing-out negative frequencies and retaining the DC component. The object computes the Hilbert transform using an equiripple FIR filter.

To compute the analytic signal of a discrete-time input:

1 Define and set up your analytic signal calculation. See "Construction" on page 5-20.

2 Call `step` to compute the analytic signal according to the properties of `dsp.AnalyticSignal`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.AnalyticSignal` returns an analytic signal object, H, that computes the complex analytic signal corresponding to each channel of a real *M*-by-*N* input matrix.

`H = dsp.AnalyticSignal('`*PropertyName*`',`*PropertyValue*`,...)` returns an analytic signal object, H, with each specified property set to the specified value.

`H = dsp.AnalyticSignal(order,'`*PropertyName*`',`*PropertyValue*`,...)` returns an analytic signal object, H, with the `FilterOrder` property set to `order` and other specified properties set to the specified values.

Properties   **FilterOrder**

Filter order used to compute Hilbert transform

Specify the order of the equiripple FIR filter used in computing the Hilbert transform as an even integer scalar. The default is `100`.

FrameBasedProcessing

Process input as frames or samples

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. The default is `true`.

Methods

clone	Create analytic signal object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of analytic signal object
step	Analytic signal

Examples

Compute the analytic signal of a sinusoidal input.

```
t = (-1:0.01:1)';
x = sin(4*pi*t);
hanlytc = dsp.AnalyticSignal(200);
y = step(hanlytc,x);

subplot(2,1,1), plot(t, x);
title('Original Signal');
subplot(2,1,2), plot(t, [real(y) imag(y)]);
title('Analytic signal of the input')
legend('Real signal','Imaginary signal',...
    'Location','best');
```

dsp.AnalyticSignal

Algorithms   This object implements the algorithm, inputs, and outputs described on the Analytic Signal block reference page. The object properties correspond to the block parameters.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the `FrameBasedProcessing` property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.FFT | dsp.IFFT

Purpose   Create analytic signal object with same property values

Syntax   `C = clone(H)`

Description   `C = clone(H)` creates an `AnalyticSignal` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

dsp.AnalyticSignal.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs to the
                 step method.

                 The getNumInputs method returns a positive integer that is the number
                 of expected inputs (not counting the object itself) to the step method.
                 This value will change if you alter any properties that turn inputs on or
                 off. You must call the step method with the number of input arguments
                 equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   `N = getNumOutputs(H)`

Description   `N = getNumOutputs(H)` returns the number of outputs, `N`, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

dsp.AnalyticSignal.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the AnalyticSignal object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.AnalyticSignal.reset

Purpose   Reset internal states of analytic signal object

Syntax   `reset(H)`

Description   `reset(H)` sets the internal states of the `AnalyticSignal` object `H` to their initial values.

Purpose   Analytic signal

Syntax   Y = step(H,X)

Description   Y = step(H,X) computes the analytic signal, Y, of the *M*-by-*N* input
                   matrix X, according to the equation

$$\mathbf{Y} = \mathbf{X} + jH\{\mathbf{X}\}$$

where *j* is the imaginary unit and $H\{\mathbf{X}\}$ denotes the Hilbert transform.

When you set the FrameBasedProcessing property to false, each of the
M-by-*N* matrix elements is an independent channel. Thus, the method
computes the analytic signal for each element of X. When you set the
FrameBasedProcessing property to true, each of the *N* columns in X
contains *M* sequential time samples from an independent channel. The
method computes the analytic signal for each channel.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.ArrayVectorAdder

Purpose   Add array to vector along specified dimension

Description   The `ArrayVectorAdder` object adds an N-D array to a vector along a specified dimension. The length of the vector must equal the size of the N-D array along the specified dimension.

To add an N-D array to a vector along a specified dimension:

1 Define and set up your array-vector addition object. See "Construction" on page 5-30.

2 Call `step` to add the N-D array according to the properties of `dsp.ArrayVectorAdder`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.ArrayVectorAdder` returns an array-vector addition object, `H`, that adds a vector to an N-D array along the first dimension.

`H = dsp.ArrayVectorAdder('`*PropertyName*`',`*PropertyValue*`, ...)` returns an array-vector addition object, `H`, with each property set to the specified value.

Properties   **Dimension**

Dimension along which to add vector elements to input

Specify the dimension along which to add the input array to the elements of the vector as a positive integer. The length of the vector must match the size of the N-D array along the specified dimension. The default is `1`.

VectorSource

Source of vector

Specify the source of the vector values as |`Input port` | `Property` |. The default is `Input port`.

Vector

Vector values

Specify the vector values. This property applies only when you set the VectorSource property to Property. The default is [0.5 0.25]. This property is tunable.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set FullPrecisionOverride to false, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as | Wrap | Saturate |. The default is Wrap. This property applies only if the object is not in full precision mode.

VectorDataType

Vector word and fraction lengths

Specify the vector fixed-point data type as | Same word length as input | Custom |. This property applies when you set the VectorSource property to Property. The default is Same word length as input.

CustomVectorDataType

Vector word and fraction lengths

Specify the vector fixed-point type as a numerictype object with a Signedness of Auto. This property applies when you set the VectorSource property to Property and the VectorDataType property to Custom. The default is numerictype([],16,15).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as | Full precision | Same as first input | Custom |. The default is Full precision.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when the AccumulatorDataType property is Custom. The default is numerictype([],32,30).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as | Same as accumulator | Same as first input | Custom |.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when the OutputDataType property is Custom. The default is numerictype([],16,15).

Methods

clone	Create array-vector adder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Add vector to N-D array

Examples   Add 2-by-1 vector to 2-by-2 matrix along the first dimension of the array.

```
hava = dsp.ArrayVectorAdder;
a = ones(2);
x = [1 2]';
y = step(hava, a, x);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Array-Vector Add block reference page. The object properties correspond to the block parameters, except:

The array-vector addition object does not have **Minimum** or **Maximum** options for data output.

See Also   dsp.ArrayVectorMultiplier | dsp.ArrayVectorDivider | dsp.ArrayVectorSubtractor

dsp.ArrayVectorAdder.clone

Purpose   Create array-vector adder object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates an array-vector adder object, C, with the same
                 property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   `N = getNumInputs(H)`

Description   `N = getNumInputs(H)` returns the number of expected inputs, N, to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(`*H*`)`.

dsp.ArrayVectorAdder.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   `isLocked(H)`

Description   `isLocked(H)` returns the locked state of the array-vector adder.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Add vector to N-D array

Syntax   Y = step(H,A)
                Y = step(H,A,V)

Description Y = step(H,A) returns Y , the result of adding the input array A to
                the elements of the vector specified in the Vector property along the
                specified dimension when the VectorSource property is Property. The
                length of the vector specified in the Vector property must equal the
                length of the specified dimension of A.

                Y = step(H,A,V) returns Y , the result of adding the input array A to
                the elements of the input vector V along the specified dimension when
                the VectorSource property is Input port. The length of the input V
                must equal the length of the specified dimension of A.

                Note The object performs an initialization the first time the step
                method is executed. This initialization locks nontunable properties and
                input specifications, such as dimensions, complexity, and data type
                of the input data. If you change a nontunable property or an input
                specification, the System object issues an error. To change nontunable
                properties or inputs, you must first call the release method to unlock
                the object.

dsp.ArrayVectorDivider

Purpose   Divide array by vector along specified dimension

Description   The ArrayVectorDivider object divides an array by a vector along a specified dimension.

To divide an array by a vector along a specified dimension:

1 Define and set up your array-vector division object. See "Construction" on page 5-40.

2 Call step to divide the array according to the properties of dsp.ArrayVectorDivider. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.ArrayVectorDivider returns an array-vector division object, H, that divides an input array by the elements of a vector along the first dimension of the array.

```
H =
dsp.ArrayVectorDivider('PropertyName',PropertyValue,...)
```
returns an array-vector division object, H, with each property set to the specified value.

Properties   **Dimension**

Dimension along which to divide input by vector elements

Specify the dimension along which to divide the input array by the elements of a vector as a positive integer. The default is 1.

VectorSource

Source of vector

Specify the source of the vector values as | Input port | Property |. The default is Input port.

Vector

Vector values

Specify the vector values. This property applies when you set the VectorSource property to Property. The default is [0.5 0.25]. This property is tunable.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as | Wrap | Saturate |. The default is Wrap.

VectorDataType

Vector word and fraction lengths

Specify the vector fixed-point mode as | Same word length as input | Custom |. This property applies when you set the VectorSource property to Property. The default is Same word length as input.

CustomVectorDataType

Vector word and fraction lengths

Specify the vector fixed-point data type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the VectorSource property to Property and the VectorDataType property to Custom. The default is numerictype([],16,15).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as | Same as first input | Custom |. The default is Same as first input.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the OutputDataType property to Custom. The default is numerictype([],16,15).

Methods		
	clone	Create array-vector division object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	step	Divide array by vector

Examples   Divide a matrix by a vector.

```
havd = dsp.ArrayVectorDivider;
a = ones(2);
x = [2 3]';
y = step(havd, a, x);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Array-Vector Divide block reference page. The object properties correspond to the block parameters, except:

The array-vector division object does not have **Minimum** or **Maximum** options for data output.

See Also   dsp.ArrayVectorMultiplier | dsp.ArrayVectorAdder | dsp.ArrayVectorSubtractor

dsp.ArrayVectorDivider.clone

Purpose   Create array-vector division object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an array-vector division object, C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

dsp.ArrayVectorDivider.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the array-vector divider.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.ArrayVectorDivider.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Divide array by vector

Syntax   Y = step(H,A,V)
                Y = step(H,A)

Description Y = step(H,A,V) returns Y , the result of dividing the input array A by
                the elements of input vector V along the specified dimension when the
                VectorSource property is Input port. The length of the input V must
                equal the length of the specified dimension of A.

                Y = step(H,A) returns Y , the result of dividing the input array A by
                the elements of the vector specified in the Vector property along the
                specified dimension when the VectorSource property is Property. The
                length of the vector specified in the Vector property must equal the
                length of the specified dimension of A.

                Note The object performs an initialization the first time the step
                method is executed. This initialization locks nontunable properties and
                input specifications, such as dimensions, complexity, and data type
                of the input data. If you change a nontunable property or an input
                specification, the System object issues an error. To change nontunable
                properties or inputs, you must first call the release method to unlock
                the object.

                ---

dsp.ArrayVectorMultiplier

Purpose

Multiply array by vector along specified dimension

Description

The `ArrayVectorMultiplier` object multiplies an array by a vector along a specified dimension.

To multiply an array by a vector along a specified:

1 Define and set up your array-vector multiplication object. See "Construction" on page 5-50.

2 Call `step` to multiply the array according to the properties of `dsp.ArrayVectorMultiplier`. The behavior of `step` is specific to each object in the toolbox.

Construction

`H = dsp.ArrayVectorMultiplier` returns an array-vector multiplication object, `H`, that multiplies an input N-D array by the elements of a vector along the second dimension.

`H = dsp.ArrayVectorMultiplier('`*PropertyName*`',`*PropertyValue*`,...)` returns an array-vector multiplication object, `H`, with each property set to the specified value.

Properties

Dimension

Dimension along which to multiply input by vector elements

Specify the dimension along which to multiply the input array by the elements of vector as a positive integer. The default is 2.

VectorSource

Source of vector

Specify the source of the vector values as one of `Input port` or `Property`. The default is `Input port`.

Vector

Vector to multiply array

Specify the vector by which to multiply the array. This property applies when you set the VectorSource property to Property. The default is [0.5 0.25]. This property is tunable.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as Ceiling, Convergent, Floor , Nearest, Round, Simplest', Zero. The default is floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as Wrap or Saturate. The default is Wrap.

VectorDataType

Vector word and fraction lengths

Specify the vector fixed-point data type as Same word length as input , Custom. This property applies when you set the VectorSource property to Property. The default is Same word length as input.

CustomVectorDataType

Vector word and fraction lengths

Specify the vector fixed-point type as a numerictype object with a Signedness of Auto. This property applies when you set the VectorSource property to Property and the VectorDataType property to Custom. The default is numerictype([],16,15).

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as `Full precision`, `Same as first input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`, `Same as product`, `Same as first input`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as product`, `Same as first input`, or `Custom`. The default is `Same as product`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when

you set the `OutputDataType` property to `Custom`. The default is
`numerictype([],16,15)`.

Methods

clone	Create array-vector multiplication object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Multiply array by vector

Examples   Multiply a matrix by a vector:

```
havm = dsp.ArrayVectorMultiplier;
a = ones(2);
x = [2 3]';
y = step(havm, a, x);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on
the Array-Vector Multiply block reference page. The object properties
correspond to the block parameters, except:

• The array-vector multiplication object does not have **Minimum** or
Maximum options for data output.

See Also   dsp.ArrayVectorAdder | dsp.ArrayVectorDivider |
dsp.ArrayVectorSubtractor

dsp.ArrayVectorMultiplier.clone

Purpose   Create array-vector multiplication object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates an array-vector multiplication object, C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                 the step method.

                 The getNumInputs method returns a positive integer that is the number
                 of expected inputs (not counting the object itself) to the step method.
                 This value will change if you alter any properties that turn inputs on or
                 off. You must call the step method with the number of input arguments
                 equal to the result of getNumInputs(*H*).

dsp.ArrayVectorMultiplier.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of output arguments from the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   `isLocked(H)`

Description   `isLocked(H)` returns the locked state of the array-vector multiplication object.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

dsp.ArrayVectorMultiplier.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Multiply array by vector

Syntax   Y = step(H,A,V)
                   Y = step(H,A)

Description   Y = step(H,A,V) returns Y , the result of multiplying the input array A
                   by the elements of input vector V along the specified dimension when
                   the VectorSource property is Input port. The length of the input V
                   must equal the length of the specified dimension of A.

                   Y = step(H,A) returns Y , the result of multiplying the input array A by
                   the elements of vector specified in Vector property along the specified
                   dimension when the VectorSource property is set to Property. The
                   length of the vector specified in Vector property must equal the length
                   of the specified dimension of A.

                   Note The object performs an initialization the first time the step
                   method is executed. This initialization locks nontunable properties and
                   input specifications, such as dimensions, complexity, and data type
                   of the input data. If you change a nontunable property or an input
                   specification, the System object issues an error. To change nontunable
                   properties or inputs, you must first call the release method to unlock
                   the object.

                   ---

dsp.ArrayVectorSubtractor

Purpose	Subtract vector from array along specified dimension
Description	The `ArrayVectorSubtractor` object subtracts a vector from an N-D array along a specified dimension.

To subtract a vector from an N-D array along a specified dimension:

1 Define and set up your array-vector subtraction object. See "Construction" on page 5-60.

2 Call `step` to subtract the vector according to the properties of `dsp.ArrayVectorSubtractor`. The behavior of `step` is specified to each object in the toolbox.

Construction

`H = dsp.ArrayVectorSubtractor` returns an array-vector subtraction object, `H`, that subtracts the elements of a vector from an N-D input array along the first dimension.

```
H =
dsp.ArrayVectorSubtractor('PropertyName',PropertyValue,...)
```
returns an array-vector subtraction object, `H`, with each property set to the specified value.

Properties

Dimension

Dimension along which to subtract vector elements from input

Specify the dimension along which to subtract the elements of the vector from the input array as an integer-valued scalar greater than `0`. The default is `1`.

VectorSource

Source of vector

Specify the source of the vector values as one of `Input port` or `Property`. The default is `Input port`.

Vector

Vector values

Specify the vector values. This property applies when you set the VectorSource property to Property. The default is [0.5 0.25]. This property is tunable.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set FullPrecisionOverride to false, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as Ceiling, Convergent, Floor, Nearest, Round, Simplest, or Zero. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of Wrap or Saturate. The default is Wrap. This property applies only if the object is not in full precision mode.

VectorDataType

Vector word and fraction lengths

Specify the vector fixed-point data type as `Same word length as input` or `Custom`. This property applies when you set the `VectorSource` property to `Property`. The default is `Same word length as input`.

CustomVectorDataType

Vector word and fraction lengths

Specify the vector fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `VectorSource` property to `Property` and the `VectorDataType` property to `Custom`. The default is `numerictype([],16,15)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Full precision` , `Same as first input`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as accumulator`, `Same as first input`, or `Custom`. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

Methods

clone	Create array-vector subtractor with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Subtract vector from array along specified dimension

Examples

Subtract a vector from a matrix:

```
havs = dsp.ArrayVectorSubtractor;
a = ones(2);
x = [1 2]';
y = step(havs, a, x);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Array-Vector Subtract block reference page. The object properties correspond to the block parameters, except:

The array-vector subtraction object does not have **Minimum** or **Maximum** options for data output.

See Also

dsp.ArrayVectorMultiplier | dsp.ArrayVectorDivider | dsp.ArrayVectorAdder

dsp.ArrayVectorSubtractor.clone

Purpose   Create array-vector subtractor with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an array-vector subtractor object, C, with the
                   same property values as H. The clone method creates a new unlocked
                   object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.ArrayVectorSubtractor.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the array-vector subtractor object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.ArrayVectorSubtractor.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

> **Note** You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose Subtract vector from array along specified dimension

Syntax Y = step(H,A,V)
Y = step(H,A)

Description Y = step(H,A,V) returns Y. The value of Y results from subtracting the elements of input vector V from the input array A along the specified dimension when the VectorSource property is Input port. The length of the input V must equal the length of the specified dimension of A.

Y = step(H,A) returns Y. The value of Y results from subtracting the elements of the vector specified in the Vector property from the input array A along the specified dimension when the VectorSource property is Property. The length of the vector specified in the Vector property must equal the length of the specified dimension of A.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

dsp.AudioFileReader

| **Purpose** | Read audio samples from audio file |

Description   The AudioFileReader object reads audio samples from an audio file.

To read audio samples from an audio file:

1 Define and set up your audio file reader object. See "Construction" on page 5-70.

2 Call step to read audio samples according to the properties of dsp.AudioFileReader. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.AudioFileReader returns an audio file reader System object, H that reads audio from an audio file. This audio file can be of the following formats on all platforms:

- MP3
- M4a
- MP4
- WAV
- FLAC
- OGG

H = dsp.AudioFileReader('*PropertyName*',*PropertyValue*,...) returns an audio file reader System object, H, with each specified property set to the specified value.

H = dsp.AudioFileReader(Filename,'*PropertyName*',*PropertyValue*,...) returns an audio file reader object, H, with Filename property set to *FILENAME* and other specified properties set to the specified values.

Properties   **Filename**

Name of audio file from which to read

Specify the name of an audio file as a string. Specify the full path for the file only if the file is not on the MATLAB path. On UNIX platforms, the audio file reader only supports uncompressed AVI files. The default is `speech_dft.avi`.

PlayCount

Number of times to play file

Specify a positive integer or `inf` as the number of times to play the file. If you enter `inf`, the file plays repeatedly. The default is `inf`.

SampleRate

Sampling rate of the audio file

This read-only property displays the sampling rate, in hertz, of the audio file.

SamplesPerFrame

Number of samples in audio frame

Specify the number of samples in an audio frame as a positive, scalar integer value. The default value is `1024`.

OutputDataType

Data type of output

Set the data type of the audio data output from the audio file reader object. Specify the data type as |double| single | int16 | uint8|. The default is `int16`.

Methods		
	clone	Create audio file reader object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method

dsp.AudioFileReader

info	Information about specific audio file
isDone	End-of-file status (logical)
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of audio file reader to read from beginning of file
step	Read audio samples from audio file

Examples

Read and play back an audio file using the standard audio output device.

```
hmfr = dsp.AudioFileReader('speech_dft.avi');
hap = dsp.AudioPlayer('SampleRate', 22050);

while ~isDone(hmfr)
   audio = step(hmfr);
   step(hap, audio);
end

release(hmfr); % release the input file
release(hap);  % release the audio output device
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the From Multimedia File block reference page. The object properties correspond to the block parameters, except:

- The object has no corresponding property for the **Inherit sample time from file** block parameter. The object always inherits the sample time from the file.

- The object has no corresponding property for the **Output end-of-file indicator** parameter. The object always outputs EOF as the last output.

- The object has no corresponding property for the **Multimedia Outputs** parameter because audio is the only supported output.

- The object has no corresponding property for the **Image signal** block parameter.

- The object has no corresponding property for the **Output color format** parameter.

- The object has no corresponding property for the **Video output data type** parameter.

See Also　　　dsp.AudioFileWriter

dsp.AudioFileReader.clone

Purpose   Create audio file reader object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an AudioFileReader System object, C, with the same property values as H.

The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of inputs, N, to the step
                  method.

                  The getNumInputs method returns a positive integer that is the number
                  of expected inputs (not counting the object itself) to the step method.
                  This value will change if you alter any properties that turn inputs on or
                  off. You must call the step method with the number of input arguments
                  equal to the result of getNumInputs(*H*).

dsp.AudioFileReader.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Information about specific audio file

Syntax   S = info(H)

Description   S = info(H) returns a MATLAB structure, S, with information about
                   the audio file specified in the Filename property. The number of fields
                   in S varies depending on the audio content of the file. For possible fields
                   and values for the structure S, see the following table.

Field	Value
SampleRate	Audio sampling rate of the audio file in Hz.
NumBits	Number of bits used to encode the audio stream.
NumChannels	Number of audio channels.

dsp.AudioFileReader.isDone

Purpose   End-of-file status (logical)

Syntax   STATUS = isDone(H)

Description STATUS = isDone(H) returns a logical value, STATUS , when the
AudioFileReader object, H , reaches the end of the audio file. If you set
the PlayCount property to a value greater than 1 , STATUS is true each
time the object reaches the end of the file. STATUS is the same as the
EOF output value in the step method syntax.

Purpose   Locked status for input attributes and nontunable properties

Syntax   y = isLocked(H)

Description   y = isLocked(H) returns the locked state, Y, of the AudioFileReader
                   object.

                   The isLocked method returns a logical value that indicates whether
                   input attributes and nontunable properties for the object are locked. The
                   object performs an internal initialization the first time the step method
                   is executed. This initialization locks nontunable properties and input
                   specifications, such as dimensions, complexity, and data type of the
                   input data. After locking, the isLocked method returns a true value.

dsp.AudioFileReader.reset

Purpose   Reset internal states of audio file reader to read from beginning of file

Syntax   reset(H)

Description   reset(H) resets the AudioFileReader object to read from the beginning
of the file.

Purpose　　　Allow property value and input characteristics changes

Syntax　　　`release(H)`

Description　　`release(H)` releases system resources, such as memory, file handles, and hardware connections. This method lets you change any properties or input characteristics.

Note　You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.AudioFileReader.step

Purpose   Read audio samples from audio file

Syntax   AUDIO = step(H)
                [AUDIO,EOF] = step(H)

Description AUDIO = step(H) outputs one frame of audio samples, AUDIO. You
                can specify the number of times to play the file using the PlayCount
                property. After playing the file for the number of times you specify,
                AUDIO contains silence.

                [AUDIO,EOF] = step(H) returns an end-of-file indicator, EOF. EOF is
                true each time the output AUDIO contains the last audio sample in the
                file.

                Note The object performs an initialization the first time the step
                method is executed. This initialization locks nontunable properties and
                input specifications, such as dimensions, complexity, and data type
                of the input data. If you change a nontunable property or an input
                specification, the System object issues an error. To change nontunable
                properties or inputs, you must first call the release method to unlock
                the object.

Purpose   Write audio samples to audio file

Description   The `AudioFileWriter` object writes audio samples to an audio file.

To write audio samples to an audio file:

1 Define and set up your audio file writer object. See "Construction" on page 5-83.

2 Call `step` to write audio samples according to the properties of `dsp.AudioFileWriter`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.AudioFileWriter` returns an audio file writer System object, `H`. This object writes audio samples to an audio file. This audio file can be of the following formats on all platforms:

- WAV
- FLAC
- OGG

`H = dsp.AudioFileWriter('`*PropertyName*`',`*PropertyValue*`,...)` returns an audio file writer System object, `H`, with each specified property set to the specified value.

`H = dsp.AudioFileWriter(`*FILENAME*`,'`*PropertyName*`',`*PropertyValue*`,...)` returns an audio file writer System object, `H`. This object has the Filename property set to *FILENAME* and other specified properties set to the specified values.

Properties   **Filename**

Name of audio file to which to write

Specify the name of the audio file as a string. The default is `output.avi`.

FileFormat

Audio file format

Specify which audio file format the object writes. On Microsoft platforms, select AVI, WAV, or WMA. On all other platforms, the only selection available is AVI. These abbreviations correspond to the following file formats:

- AVI: Audio-Video Interleave

- WAV: Microsoft WAVE Files

- WMA: Windows Media Audio

The default is AVI.

SampleRate

Sampling rate of audio data stream

Specify the sampling rate of the input audio data as a positive, numeric scalar value. The default is 44100 .

Compressor

Algorithm that compresses audio data

Specify the type of compression algorithm the audio file writer uses to compress the audio data. Compression reduces the size of the audio file. Select None (uncompressed) to save uncompressed audio data to the file. The other options available reflect the audio compression algorithms installed on your system. You can use tab completion to query valid Compressor options for your computer by typing H.Compressor = ' and then pressing the tab key. This property applies when writing WAV or AVI files on Windows platforms.

DataType

Data type of the uncompressed audio

Specify the type of uncompressed audio data written to the file as one of inherit, int16, int24, single, or uint8. This property only applies when writing uncompressed WAV files. The default value of this property is int16.

Methods

clone	Create audio file writer object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Write one frame of audio output samples

Examples

Decimate an audio signal, and write it to disk as an AVI file.

```
hmfr = dsp.AudioFileReader('OutputDataType',...
   'double');
hfirdec = dsp.FIRDecimator; % decimate by 2
hmfw = dsp.AudioFileWriter...
   ('speech_dft_11025.avi', ...
   'SampleRate', 22050/2);

while ~isDone(hmfr)
    audio = step(hmfr);
    audiod = step(hfirdec, audio);
    step(hmfw, audiod);
end

release(hmfr);
```

```
release(hmfw);
```

Algorithms   This object implements the algorithm, inputs, and outputs described
on the To Multimedia File block reference page. The object properties
correspond to the block parameters, except:

- The object `FileFormat` property does not support video-only file
  formats.

- The object has no corresponding property for the **Write** parameter.
  The object writes only audio content to files.

- The object has no corresponding property for the **Video compressor**
  parameter.

- The object has no corresponding property for the **File color format**
  parameter.

- The object has no corresponding property for the **Image signal**
  parameter.

See Also   `dsp.AudioFileReader`

Purpose   Create audio file writer object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an AudioFileWriter System object C, with the same property values as H. The clone method creates a new unlocked object.

dsp.AudioFileWriter.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, of
the step method

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, of step methodN, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

dsp.AudioFileWriter.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the `AudioFileWriter` System object H.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections. This method lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Write one frame of audio output samples

Syntax   `step(H,AUDIO)`

Description `step(H,AUDIO)` writes one frame of audio samples, `AUDIO`, to the output file. `AUDIO` is either a vector or an M-by-N matrix for mono or N-channel audio inputs respectively.

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose   Play audio data using computer's audio device

Description   The AudioPlayer object plays audio data using the computer's audio device.

To play audio data using the computer's audio device:

1 Define and set up your audio player object. See "Construction" on page 5-93.

2 Call step to play audio data according to the properties of dsp.AudioPlayer. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.AudioPlayer returns an audio player object, H, that plays audio samples using an audio output device in real-time.

H = dsp.AudioPlayer('*PropertyName*',*PropertyValue*, ...) returns an audio player object, H, with each property set to the specified value.

H = dsp.AudioPlayer(SAMPLERATE,'*PropertyName*',*PropertyValue*, ...) returns an audio player object, H, with the SampleRate property set to SAMPLERATE and other specified properties set to the specified values.

Properties   **DeviceName**

Device to which to send audio data

Specify the device to which to send the audio data. The default is Default, which is the computer's standard output device. You can use tab completion to query valid DeviceName assignments for your computer by typing H.DeviceName = ' and then pressing the tab key.

SampleRate

Number of samples per second sent to audio device

Specify the number of samples per second in the signal as an integer. The default is 44100.

DeviceDataType

Data type used by device

Specify the data type used by the audio device to acquire audio data as `Determine from input data type`, `8-bit integer`, `16-bit integer`, `24-bit integer`, or `32-bit float`. The default is `Determine from input data type`.

BufferSizeSource

Source of Buffer Size

Specify how to determine the buffer size as `Auto` or `Property`. The default is `Auto`.

BufferSize

Buffer size

Specify the size of the buffer that the audio player object uses to communicate with the audio device as an integer. This property applies when you set the `BufferSizeSource` property to `Property`. The default is `4096`.

QueueDuration

Size of queue in seconds

Specify the length of the audio queue, in seconds. The default is `1.0`.

Methods	clone	Create audio player object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Write audio to audio output device

Examples Read in an AVI audio file, and play the file back using the standard audio output device:

```
hafr = dsp.AudioFileReader;
hap = dsp.AudioPlayer('SampleRate',22050);

while ~isDone(hafr)
   audio = step(hafr);
   step(hap,audio);
end

pause(hap.QueueDuration);  % Wait until audio plays to the end

release(hafr); % close input file, release resources
release(hap);  % close audio output device, release resources
```

Algorithms This object implements the algorithm, inputs, and outputs described on the To Audio Device block reference page. The object properties correspond to the block parameters.

See Also dsp.AudioFileReader | dsp.AudioRecorder

How To • Set the Audio Hardware API

dsp.AudioPlayer.clone

Purpose   Create audio player object with same property values

Syntax   `C = clone(H)`

Description   `C = clone(H)` creates an audio player object, `C`, with the same property values as `H`. The clone method creates a new unlocked object.

Purpose Number of expected inputs to step method

Syntax N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

dsp.AudioPlayer.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step
                method.

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the audio player object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.AudioPlayer.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Write audio to audio output device

Syntax   Y = step(H,AUDIO)

Description   Y = step(H,AUDIO) writes one frame of AUDIO samples to the selected
                   audio output device, Y.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.AudioRecorder

Purpose   Record audio data using computer's audio device

Description   The AudioRecorder object records audio data using the computer's audio device.

To record audio data using the computer's audio device:

1 Define and set up your audio recorder object. See "Construction" on page 5-102.

2 Call step to record audio data according to the properties of dsp.AudioRecorder. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.AudioRecorder returns an audio recorder object, H, that records audio samples using an audio input device in real-time.

H = dsp.AudioRecorder('*PropertyName*',*PropertyValue*, ...) returns an audio recorder object, H, with each property set to the specified value.

Properties   **DeviceName**

Device from which to acquire audio data

Specify the device from which to acquire audio data. The default is Default, which is the computer's standard input device. You can use tab completion to query valid DeviceName assignments for your computer. by typing H.DeviceName = ' and then pressing the tab key. The tab completion functionality shows all valid audio device names for your computer.

SampleRate

Number of samples per second read from audio device

Specify the number of samples per second in the signal as an integer. The default is 44100.

NumChannels

Number of audio channels

Specify the number of audio channels as an integer. The default is 2.

DeviceDataType

Data type used by device

Specify the data type used by the device to acquire audio data as `Determine from output data type`, `8-bit integer`, `16-bit integer`, `24-bit integer`, or `32-bit float`. The default is `Determine from output data type`.

BufferSizeSource

Source of Buffer Size

Specify how to determine the buffer size as `Auto` or `Property`. The default is `Auto`.

BufferSize

Buffer size

Specify as an integer the size of the buffer that the audio recorder object uses to communicate with the audio device. This property applies when you set the `BufferSizeSource` property to `Property`. The default is `4096`.

QueueDuration

Size of queue in seconds

Specify the length of the audio queue, in seconds. The default is 1.0.

SamplesPerFrame

Number of samples in the output signal

Specify the number of samples in the audio recorder's output as an integer. The default is 1024.

OutputDataType

Data type of the output

Select the output data type as `uint8`, `int16`, `int32` , `single` , or `double`. The default is `single`.

Methods

clone	Create audio recorder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Record audio from computer's recording device

Examples

Record ten seconds of speech from a microphone, and send the output to a .wav file:

```
har = dsp.AudioRecorder;
hmfw = dsp.AudioFileWriter('myspeech.avi','FileFormat','AVI');

disp('Speak into microphone now');

tic;
while toc < 10,
    step(hmfw, step(har));
end

release(har);
release(hmfw);
disp('Recording complete');
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the From Audio Device block reference page. The object properties correspond to the block parameters.

See Also   dsp.AudioFileReader | dsp.AudioPlayer

How To   • Set the Audio Hardware API

Purpose   Create audio recorder object with same property values

Syntax   `clone(H)`

Description `clone(H)` creates an audio recorder object, `C`, with the same property values as `H`. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
the step method

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.AudioRecorder.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step
method.

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the audio recorder.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.AudioRecorder.release

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Record audio from computer's recording device

Syntax   AUDIO = step(H)

Description   AUDIO = step(H) reads one frame of audio samples from the selected
                  audio input device.

dsp.Autocorrelator

Purpose   Autocorrelation sequence

Description   The `Autocorrelator` object returns the autocorrelation sequence for a discrete-time, deterministic input, or the autocorrelation sequence estimate for a discrete-time, wide-sense stationary (WSS) random process at positive lags.

To obtain the autocorrelation sequence:

1 Define and set up your autocorrelator. See "Construction" on page 5-112.

2 Call `step` to compute the autocorrelation sequence according to the properties of `dsp.Autocorrelator`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.Autocorrelator` returns an autocorrelator, `H`, that computes the autocorrelation along the first dimension of an N-D array. By default, the autocorrelator computes the autocorrelation at lags from zero to $N-1$, where N is the length of the input vector or the row dimension of the input matrix. Inputting a row vector results in a row of zero-lag autocorrelation sequence values, one for each column of the row vector. The default autocorrelator returns the unscaled autocorrelation and performs the computation in the time domain.

`H = dsp.Autocorrelator('`*PropertyName*`',`*PropertyValue*`, ...)` returns an autocorrelator, `H`, with each property set to the specified value.

Properties   **MaximumLagSource**

Source of maximum lag

Specify how to determine the range of lags for the autocorrelation as `Auto` or `Property`. If the value of `MaximumLagSource` is `Auto`, the autocorrelator computes the autocorrelation over all nonnegative lags in the interval [`0`, *N*-`1`], where *N* is the length of the first dimension of the input. Otherwise, the object computes

the autocorrelation using lags in the range [0,MaximumLag]. The default is Auto.

MaximumLag

Maximum positive lag

Specify the maximum lag as a positive integer. This property applies only when the MaximumLagSource property is Property. The MaximumLag must be less than the length of the input data. The default is 1.

Scaling

Autocorrelation function scaling

Specify the scaling to apply to the output as None, Biased, Unbiased, or Unity at zero-lag. Set this property to None to generate the autocorrelation function without scaling. This option is appropriate if you are computing the autocorrelation of a nonrandom (deterministic) input.

The Biased option scales the autocorrelation by $1/N$, where N is the length of the input data. Scaling by $1/N$ yields a biased, finite-sample approximation to the theoretical autocorrelation of a WSS random process. In spite of the bias, scaling by $1/N$ has the desirable property that the sample autocorrelation matrix is nonnegative definite, a property possessed by the theoretical autocorrelation matrices of all wide-sense stationary random processes. The Fourier transform of the biased autocorrelation estimate is the *periodogram*, a widely used estimate of the power spectral density of a WSS process.

The Unbiased option scales the estimate of the autocorrelation by 1/N-1. Scaling by $N - 1$ produces an unbiased estimate of the theoretical autocorrelation. However, using the unbiased option, you can obtain an estimate of the autocorrelation function that fails to have the nonnegative definite property.

Use the Unity at zero-lag option to normalize the autocorrelation estimate as identically one at lag zero. See

"Definitions" on page 5-116 for more detail on scaling. The default is None.

Method

Domain for computing autocorrelations

Specify the domain for computing autocorrelations as Time Domain or Frequency Domain. You must set this property to Time Domain for fixed-point signals. The default is Time Domain.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set FullPrecisionOverride to false, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as Ceiling, Convergent, Floor, Nearest, Round, Simplest, or Zero. This property applies only when you set the Method property to Time Domain. The default is Floor. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. This property applies only when you set the `Method` property to `Time Domain`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of `Full precision`, `Same as input`, or `Custom`. This property applies only when you set the `Method` property to `Time Domain`. The default is `Full precision`

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `Method` property to `Time Domain` and the `ProductDataType` property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Full precision`, `Same as product`, `Same as input`, or `Custom`. This property applies only when the `Method` property is `Time Domain`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `Method` property to `Time Domain` and

the `AccumulatorDataType` property to `Custom`. The default `isnumerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as accumulator`, `Same as product`, `Same as input`, or `Custom`. This property applies only when the `Method` property is `Time Domain`. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `Method` property to `Time Domain` and the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

Methods		
	clone	Create autocorrelator object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	step	Autocorrelation sequence

Definitions   The autocorrelation of a deterministic discrete-time sequence, *x(n)* is:

$$r_x(h) = \sum_{n=0}^{N-h-1} x^*(n)x(n+h) \quad h = 0,1,...,N-1$$

where h is the lag and $*$ denotes the complex conjugate. If the input is a length N realization of a WSS stationary random process, $r_x(h)$ is an estimate of the theoretical autocorrelation:

$$\rho_x(h) = E\{x^*(n)x(n+h)\}$$

where $E\{\}$ is the expectation operator. The `Unity at zero-lag` normalization divides each sequence value by the autocorrelation or autocorrelation estimate at zero lag.

$$\frac{\rho_x(h)}{\rho_x(0)} = \frac{E\{x^*(n)x(n+h)\}}{E\{|x(0)|^2\}}$$

The most commonly used estimate of the theoretical autocorrelation of a WSS random process is the biased estimate:

$$\hat{\rho}_x(h) = \frac{1}{N} \sum_{k=0}^{N-h-1} x^*(n)x(n+h)$$

Examples

Compute autocorrelation for all positive lags:

```
hac1 = dsp.Autocorrelator;
% x is a column vector
x = [1:100]';
y = step(hac1, x);
```

Compute the autocorrelation of a sine wave in white Gaussian noise with approximate 95% upper and lower confidence limits:

```
S = rng('default');
% Sine wave with period N=4
x = 1.4*cos(pi/2*(1:100))'+randn(100,1);
```

```
MaxLag = 20;
H = dsp.Autocorrelator('MaximumLagSource',...
'Property','MaximumLag',MaxLag,'Scaling','Unity at zero-lag');
SigAutocorr = step(H,x);
stem(SigAutocorr,'b','markerfacecolor',[0 0 1]);
line(1:MaxLag+1,1.96/sqrt(100)*ones(MaxLag+1,1),...
     'linestyle','-.','linewidth',2);
line(1:MaxLag+1,-1.96/sqrt(100)*ones(MaxLag+1,1),...
     'linestyle','-.','linewidth',2);
axis([1 20 -1 1]);
title('Sine Wave + Noise Autocorrelation'); xlabel('Lag');
```

As this figure shows, the autocorrelation estimate demonstrates the four sample periodic sine wave with excursions outside the 95% white Gaussian noise confidence limits every two samples.

dsp.Autocorrelator

Algorithms   This object implements the algorithm, inputs, and outputs described on the Autocorrelation block reference page. The object properties correspond to the block parameters.

See Also   dsp.Crosscorrelator

Purpose   Create autocorrelator object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an autocorrelator object, C, with the same
                   property values as H. The clone method creates a new unlocked object.

dsp.Autocorrelator.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description `N = getNumInputs(H)` returns the number of expected inputs, N, to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                     method

                     The getNumOutputs method returns a positive integer that is the
                     number of outputs from the step method. This value will change if you
                     alter any properties that turn outputs on or off.

dsp.Autocorrelator.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the autocorrelator.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.Autocorrelator.step

Purpose	Autocorrelation sequence
Syntax	`Y = step(H,X)`
Description	`Y = step(H,X)` computes the autocorrelation sequence `Y` for the columns of the input `X`.

> **Note** The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose   IIR filter using biquadratic structures

Description   The BiquadFilter object implements an IIR filter structure using biquadratic or second–order sections (SOS).

To implement an IIR filter structure using biquadratic or SOS:

1 Define and set up your biquadratic IIR filter. See "Construction" on page 5-127.

2 Call step to implement the IIR filter according to the properties of dsp.BiquadFilter. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.BiquadFilter returns a default biquadratic IIR filter, H, which independently filters each channel of the input over successive calls to the step method using the SOS section[1 0.3 0.4 1 0.1 0.2] with a direct-form II transposed structure.

H = dsp.BiquadFilter('*PropertyName*',*PropertyValue*, ...) returns a biquadratic filter object, H, with each property set to the specified value.

Properties   **Structure**

Filter structure

Specify the filter structure as one of | Direct form I | Direct form I transposed | Direct form II | Direct form II transposed |. The default is Direct form II transposed.

SOSMatrixSource

SOS matrix source

Specify the source of the SOS matrix as one of | Property | Input port|. The default is Property.

SOSMatrix

SOS matrix

Specify the second-order section (SOS) matrix as an N-by-6 matrix, where N is the number of sections in the filter. The default is [1 0.3 0.4 1 0.1 0.2]. Each row of the SOS matrix contains the numerator and denominator coefficients of the corresponding section of the filter. The system function, *H(z)*, of a biquad filter is:

$$H(z) = \frac{\displaystyle\sum_{k=0}^{2} b_k z^{-k}}{1 - \displaystyle\sum_{l=1}^{2} a_l z^{-l}}$$

The coefficients are ordered in the rows of the SOS matrix as (b_0, b_1 b_2 1, $-a_1$, $-a_2$). You can use coefficients of real or complex values. This property applies only when you set the SOSMatrixSource property to Property. The leading denominator coefficient of the biquad filter, a_0, equals 1 for each filter section, regardless of the specified value. This property is tunable when the OptimizeUnityScaleValues property is false.

ScaleValues

Scale values for each biquad section

Specify the scale values to apply before and after each section of a biquad filter. ScaleValues must be either a scalar or a vector of length N+1, where N is the number of sections. If you set this property to a scalar, the scalar value is used as the gain value only before the first filter section. The remaining gain values are set to 1. If you set this property to a vector of N+1 values, each value is used for a separate section of the filter.

This property applies only when you set the SOSMatrixSource property to Property. The default is 1. This property is tunable when the OptimizeUnityScaleValues property is false.

InitialConditions

Initial conditions for direct form II structures

Specify the initial conditions of the filter states when the
Structure property is one of | Direct form II | Direct form
II transposed |. The number of states or delay elements (zeros
and poles) in a direct-form II biquad filter equals twice the
number of filter sections. You can specify the initial conditions
as a scalar, vector, or matrix.

When you specify a scalar value, the biquad filter initializes all
delay elements in the filter to that value. When you specify a
vector of length equal to the number of delay elements in the
filter, each vector element specifies a unique initial condition for
the corresponding delay element.

The biquad filter applies the same vector of initial conditions to
each channel of the input signal. When you specify a vector of
length equal to the product of the number of input channels and
the number of delay elements in the filter, each element specifies
a unique initial condition for the corresponding delay element in
the corresponding channel. When you specify a matrix with the
same number of rows as the number of delay elements in the filter,
and one column for each channel of the input signal, each element
specifies a unique initial condition for the corresponding delay
element in the corresponding channel. The default is the scalar 0.

NumeratorInitialConditions

Initial conditions on zeros side

Specify the initial conditions of the filter states on the side of the
filter structure with the zeros. This property applies only when
you set the Structure property to one of | Direct form I |
Direct form I transposed |. The number of states or delay
elements in the numerator of a direct-form I biquad filter equals
twice the number of filter sections. You can specify the initial
conditions as a scalar, vector, or matrix. When you specify a
scalar, the biquad filter initializes all delay elements on the zeros
side in the filter to that value. When you specify a vector of length
equal to the number of delay elements on the zeros side in the

filter, each vector element specifies a unique initial condition for the corresponding delay element on the zeros side.

The biquad filter applies the same vector of initial conditions to each channel of the input signal. When you specify a vector of length equal to the product of the number of input channels and the number of delay elements on the zeros side in the filter, each element specifies a unique initial condition for the corresponding delay element on the zeros side in the corresponding channel. When you specify a matrix with the same number of rows as the number of delay elements on the zeros side in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element on the zeros side in the corresponding channel. The default is the scalar `0`.

DenominatorInitialConditions

Initial conditions on poles side

Specify the initial conditions of the filter states on the side of the filter structure with the poles. This property only applies when you set the `Structure` property to one of | `Direct form I` | `Direct form I transposed` |. The number of denominator states, or delay elements, in a direct-form I (noncanonic) biquad filter equals twice the number of filter sections. You can specify the initial conditions as a scalar, vector, or matrix. When you specify a scalar, the biquad filter initializes all delay elements on the poles side of the filter to that value. When you specify a vector of length equal to the number of delay elements on the poles side in the filter, each vector element specifies a unique initial condition for the corresponding delay element on the poles side.

The object applies the same vector of initial conditions to each channel of the input signal. When you specify a vector of length equal to the product of the number of input channels and the number of delay elements on the poles side in the filter, each element specifies a unique initial condition for the corresponding delay element on the poles side in the corresponding channel.

When you specify a matrix with the same number of rows as the number of delay elements on the poles side in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element on the poles side in the corresponding channel. The default is the scalar `0`.

OptimizeUnityScaleValues

Optimize unity scale values

When this Boolean property is set to `true`, the biquad filter removes all unity scale gain computations. This reduces the number of computations and increases the fixed-point accuracy. This property applies only when you set the `SOSMatrixSource` property to `Property`. The default is `true`.

ScaleValuesInputPort

How to specify scale values

Select how to specify scale values. This property applies only when the `SOSMatrixSource` property is `Input port`. By default, this property is `true`, and the scale values are specified via the input port. When this property is `false`, all scale values are 1.

FrameBasedProcessing

Enable frame-based processing

Set this property to `true` to enable frame-based processing. When this property is `true`, the biquad filter treats each column as an independent channel. Set this property to `false` to enable sample-based processing. When this property is `false`, the biquad filter treats each element of the input as an individual channel. The default is `true`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap.

MultiplicandDataType

Multiplicand word and fraction lengths

Specify the multiplicand fixed-point data type as one of | Same as output | Custom |. This property applies only when you set the Structure property to Direct form I transposed. The default is Same as output.

CustomMultiplicandDataType

Custom multiplicand word and fraction lengths

Specify the multiplicand fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the MultiplicandDataType property to Custom. The default is numerictype([],32,30).

SectionInputDataType

Section input word and fraction lengths

Specify the section input fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomSectionInputDataType

Custom section input word and fraction lengths

Specify the section input fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the SectionInputDataType property to Custom. The default is numerictype([],16,15).

SectionOutputDataType

Section output word and fraction lengths

Specify the section output fixed-point data type as one of | Same as section input | Custom |. The default is Same as section input.

CustomSectionOutputDataType

Custom section output word and fraction lengths

Specify the section output fixed-point type as a signed, scaled numerictype object with a Signedness of Auto. This property applies only when you set the SectionOutputDataType property to Custom. The default is numerictype([],16,15).

NumeratorCoefficientsDataType

Numerator coefficients word and fraction lengths

Specify the numerator coefficients fixed-point data type as one of | Same word length as input | Custom |. This property applies only when you set the SOSMatrixSource property to Property. Setting this property also sets the DenominatorCoefficientsDataType and ScaleValuesDataType properties to the same value. The default is Same word length as input.

CustomNumeratorCoefficientsDataType

Custom numerator coefficients word and fraction lengths

Specify the numerator coefficients fixed-point type as a numerictype object with a Signedness of Auto. This property applies only when you set the SOSMatrixSource property to Property and the NumeratorCoefficientsDataType property to Custom. The word length of the CustomNumeratorCoefficientsDataType, CustomDenominatorCoefficientsDataType, and CustomScaleValuesDataType properties must be the same. The default is numerictype([],16,15).

DenominatorCoefficientsDataType

Denominator coefficients word and fraction lengths

Specify the denominator coefficients fixed-point data type as one of | Same word length as input | Custom |. This property applies only when you set the SOSMatrixSource property to Property. Setting this property also sets the NumeratorCoefficientsDataType and ScaleValuesDataType properties to the same value. The default is Same word length as input.

CustomDenominatorCoefficientsDataType

Custom denominator coefficients word and fraction lengths

Specify the denominator coefficients fixed-point type as a numerictype object with a Signedness of Auto. This property applies only when you set the SOSMatrixSource property to Property and the DenominatorCoefficientsDataType property to Custom. The CustomNumeratorCoefficientsDataType, CustomDenominatorCoefficientsDataType, and CustomScaleValuesDataType properties must have the same word lengths. The default is numerictype([],16,15).

ScaleValuesDataType

Scale values word and fraction lengths

Specify the scale values fixed-point data type as one of | Same word length as input | Custom |. This property applies only when you set the SOSMatrixSource property to Property. Setting this property also sets the NumeratorCoefficientsDataType and DenominatorCoefficientsDataType properties to the same value. The default is Same word length as input.

CustomScaleValuesDataType

Custom scale values word and fraction lengths

Specify the scale values fixed-point type as a numerictype object with a Signedness of Auto. This property applies

only when you set the SOSMatrixSource property to Property and the ScaleValuesDataType property to Custom. The CustomNumeratorCoefficientsDataType, CustomDenominatorCoefficientsDataType, and CustomScaleValuesDataType properties must have the same word lengths. The default is numerictype([],16,15).

NumeratorProductDataType

Numerator product word and fraction lengths

Specify the product fixed-point data type as one of | Same as input | Custom |. Setting this property also sets the DenominatorProductDataType property to the same value. The default is Same as input.

CustomNumeratorProductDataType

Custom numerator product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the NumeratorProductDataType property to Custom. The CustomNumeratorProductDataType and CustomDenominatorProductDataType properties must have the same word lengths. The default is numerictype([],32,30).

DenominatorProductDataType

Denominator product word and fraction lengths

Specify the product fixed-point data type as one of | Same as input | Custom |. Setting this property also sets the NumeratorProductDataType property to the same value. The default is Same as input.

CustomDenominatorProductDataType

Custom denominator product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the DenominatorProductDataType

to `Custom`. The `CustomNumeratorProductDataType` and `CustomDenominatorProductDataType` properties must have the same word lengths. The default is `numerictype([],32,30)`.

NumeratorAccumulatorDataType

Numerator accumulator word and fraction lengths

Specify the numerator accumulator fixed-point data type as one of | `Same as input` | `Same as product` | `Custom` |. Setting this property also sets the `DenominatorAccumulatorDataType` property to the same value. The default is `Same as product`.

CustomNumeratorAccumulatorDataType

Custom numerator accumulator word and fraction lengths

Specify the numerator accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `NumeratorAccumulatorDataType` property to `Custom`. The `CustomNumeratorAccumulatorDataType` and `CustomDenominatorAccumulatorDataType` properties must have the same word lengths. The default is `numerictype([],32,30)`.

DenominatorAccumulatorDataType

Denominator accumulator word and fraction lengths

Specify the denominator accumulator fixed-point data type as one of | `Same as input` | `Same as product` | `Custom`|. Setting this property also sets the `NumeratorAccumulatorDataType` property to the same value. The default is `Same as product`.

CustomDenominatorAccumulatorDataType

Custom denominator accumulator word and fraction lengths

Specify the denominator accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `DenominatorAccumulatorDataType` property to `Custom`. The `CustomNumeratorAccumulatorDataType` and `CustomDenominatorAccumulatorDataType` properties

must have the same word lengths. The default is
`numerictype([],32,30)`.

StateDataType

State word and fraction lengths

Specify the state fixed-point data type as one of | `Same as input` | `Same as accumulator` | `Custom`|. This property applies when you set the `Structure` property to `Direct form II` or `Direct form II transposed`. The default is `Same as accumulator`.

CustomStateDataType

Custom state word and fraction lengths

Specify the state fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `StateDataType` property to `Custom`. The default is `numerictype([],16,15)`.

NumeratorStateDataType

Numerator state word and fraction lengths

Specify the numerator state fixed-point data type as one of | `Same as input` | `Same as accumulator` | `Custom` |. Setting this property also sets the `DenominatorStateDataType` property to the same value. This property applies only when you set the `Structure` property to `Direct form I transposed`. The default is `Same as accumulator`.

CustomNumeratorStateDataType

Custom numerator state word and fraction lengths

Specify the numerator state fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `StateDataType` property to `Custom`. The `CustomNumeratorProductDataType` and `CustomDenominatorProductDataType` properties must have the same word lengths. The default is `numerictype([],16,15)`.

DenominatorStateDataType

Denominator state word and fraction lengths

Specify the denominator state fixed-point data type as one of | Same as input | Same as accumulator | Custom |. Setting this property also sets the NumeratorStateDataType property to the same value. This property applies only when you set the Structure property to Direct form I transposed. The default is Same as accumulator.

CustomDenominatorStateDataType

Custom denominator state word and fraction lengths

Specify the denominator state fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the StateDataType property to Custom. The CustomNumeratorStateDataType and CustomDenominatorStateDataType properties must have the same word lengths. The default is numerictype([],16,15).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | Same as input | Same as accumulator | Custom |. The default is Same as accumulator.

CustomOutputDataType

Custom output word and fraction lengths

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the OutputDataType property to Custom. The default is numerictype([],16,15).

Methods

clone	Create biquad filter object with same property values
freqz	Frequency response
fvtool	Open filter visualization tool
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
impz	Impulse response
isLocked	Locked status for input attributes and nontunable properties
phasez	Unwrapped phase response
release	Allow property value and input characteristics changes
reset	Reset states of biquad filter object
step	Filter input with biquad filter object

More "Analysis Methods for Filter System Objects" on page 4-2.

Examples

Use a fourth order, lowpass biquadratic filter object with a normalized cutoff frequency of 0.4 to filter high frequencies from an input signal. Display the result as a PSD using the Welch method:

```
t = (0:1000)'/8e3;
xin = sin(2*pi*1e3*t)+sin(2*pi*3e3*t); % Input is 1 &
                                       % 3kHz sinusoids
hFromWS = dsp.SignalSource(xin, 100);
hLog = dsp.SignalSink;

[z,p,k] = ellip(4,1,60,.4);    % Set up the filter
[s,g] = zp2sos(z,p,k);
```

```
hBqF=dsp.BiquadFilter('Structure','Direct form I', ...
   'SOSMatrix',s,'ScaleValues',g);

while ~isDone(hFromWS)
     input = step(hFromWS);
     filteredOutput = step(hBqF,input);
     step(hLog,filteredOutput);
end

filteredResult = hLog.Buffer;
swelch = spectrum.welch;
psd(swelch,filteredResult,'Fs',8e3);
```

Design and apply a lowpass Butterworth filter using a biquad filter object. Create another object for Welch spectral analysis.

```
Fs = 10000; % Sampling frequency 10 kHz
t = 0:(1/Fs):1;
x = cos(2*pi*250*t)+0.5*sin(2*pi*1000*t)+0.25*cos(2*pi*2000*t);

% Design SOS Butterworth filter with fdesign.lowpass
Hd = fdesign.lowpass('Fp,Fst,Ap,Ast',500,550,0.5,60,10000);
D = design(Hd,'butter');
H = dsp.BiquadFilter('SOSMatrix',D.sosMatrix,...
'ScaleValues',D.ScaleValues);
y = step(H,x');

SpecWelch = spectrum.welch('Hamming',256,70);
InputPsd = psd(SpecWelch,x,'Fs',1e4);
OutputPsd = psd(SpecWelch,y,'Fs',1e4);

plot(InputPsd.Frequencies,10*log10(InputPsd.Data),'b');
hold on;
plot(OutputPsd.Frequencies,10*log10(OutputPsd.Data),'k',...
  'linewidth',2);
xlabel('Hz'); ylabel('dB');
legend('Input Data','Output Data','Location','best');
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Biquad Filter block reference page. The object properties correspond to the block parameters, except:

- **Coefficient source** – the biquad filter object does not accept `dfilt` objects as an `SOSMatrixSource`.

- **Action when the a0 values of the SOS matrix are not one** – the biquad filter object assumes the zero-th-order denominator coefficient equals 1 regardless of the specified value. The biquad filter object does not support the `Error` or `Warn` options found in the corresponding block.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the `FrameBasedProcessing` property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.DigitalFilter

dsp.BiquadFilter.clone

Purpose   Create biquad filter object with same property values

Syntax   `C = clone(H)`

Description   `C = clone(H)` creates a biquad filter object, `C`, with the same property values as `H`. The clone method creates a new unlocked object with uninitialized states.

Purpose   Frequency response

Syntax
```
[h,w] = freqz(H)
[h,w] = freqz(H,n)
[h,w] = freqz(H,Name,Value)
freqz(H)
```

Description   `[h,w] = freqz(H)` returns the complex, 8192–element frequency response vector h, and the corresponding frequencies w in radians/sample.

`[h,w] = freqz(H,n)` returns the complex, n-element frequency response vector h, and the corresponding frequencies w in radians/sample, using n samples.

`[h,w] = freqz(H,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(H)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object H.

dsp.BiquadFilter.fvtool

Purpose	Open filter visualization tool
Syntax	`fvtool(H)` `fvtool(H,'Arithmetic',ARITH,...)`
Description	`fvtool(H)` performs an analysis and computes the magnitude response of the filter System object `H`. `fvtool(H,'Arithmetic',ARITH,...)` analyzes the filter System object `H`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                    the step method.

                    The getNumInputs method returns a positive integer that is the number
                    of expected inputs (not counting the object itself) to the step method.
                    This value will change if you alter any properties that turn inputs on or
                    off. You must call the step method with the number of input arguments
                    equal to the result of getNumInputs(*H*).

dsp.BiquadFilter.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step
method

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

Purpose   Impulse response

Syntax   ```
[h,t] = impz(H)
[h,t] = impz(H,Name,Value)
impz(H)
```

Description `[h,t] = impz(H)` returns the impulse response h, and the
corresponding time points t at which the impulse response of H is
computed.

`[h,t] = impz(H,Name,Value)` returns the impulse response h, and the
corresponding time points t , with additional options specified by one or
more Name, Value pair arguments.

`impz(H)` uses FVTool to plot the impulse response of the filter System
object H.

Note You can use `impz` for both real and complex filters. When you
omit the output arguments, `impz` plots only the real part of the impulse
response.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the biquad filter object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Unwrapped phase response

Syntax
```
[phi,w] = phasez(H)
[phi,w] = phasez(H,n)
[phi,w] = phasez(H,Name,Value)
phasez(H)
```

Description   `[phi,w] = phasez(H)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(H,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using n samples.

`[phi,w] = phasez(H,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(H)` uses FVTool to plot the phase response of the filter System object `H`.

dsp.BiquadFilter.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset states of biquad filter object

Syntax   `reset(H)`

Description   `reset(H)` resets the filter states of the biquad filter object,`H`, to the specified initial conditions. After the `step` method applies the biquad filter object to nonzero input data, the states may change. Invoking the `step` method again without first invoking the `reset` method may produce different outputs for an identical input.

Example of resetting filter states:

```
n =0:20;
x = cos(pi/4*n)+sin(pi/2*n);
H = dsp.BiquadFilter;
y = step(H,x');
% Apply step method without invoking reset
y1 = step(H,x');
isequal(y,y1) % Returns a 0
% Reset filter states to zero
reset(H)
% Now invoke step method
y2 = step(H,x');
isequal(y,y2) % Returns a 1
```

Purpose   Filter input with biquad filter object

Syntax
```
Y = step(H,X)
Y = step(H,X,NUM,DEN)
Y = step(H,X,NUM,DEN,G)
```

Description   `Y = step(H,X)` filters the real or complex input signal `X` , and outputs the filtered values, `Y`. The biquad filter object filters each channel of the input signal over successive calls to the `step` method.

`Y = step(H,X,NUM,DEN)` filters the input using `NUM` as the numerator coefficients, and `DEN` as the denominator coefficients of the biquad filter. `NUM` must be a 3-by-N numeric matrix and `DEN` must be a 2-by-N numeric matrix, where N is the number of biquad filter sections. The object assumes that the first denominator coefficient of each section is 1. This configuration applies when the `SOSMatrixSource` property is `Input port` and the `ScaleValuesInputPort` property is `false`.

`Y = step(H,X,NUM,DEN,G)` specifies the scale values, `G`, of the biquad filter. `G` must be a 1-by-(N +1) numeric vector, where N is the number of biquad filter sections. This configuration applies when the `SOSMatrixSource` property is `Input Port` and the `ScaleValuesInputPort` property is true.

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose   Output, error, and weights using Block LMS adaptive algorithm

Description   The `BlockLMSFilter` object computes output, error, and weights using the Block LMS adaptive algorithm.

To compute the output, error, and weights:

1 Define and set up your adaptive FIR filter. See "Construction" on page 5-153.

2 Call `step` to compute the output, error, and weights according to the properties of `dsp.BlockLMSFilter`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H=dsp.BlockLMSFilter` returns an adaptive FIR filter, `H`, that filters the input signal and computes filter weights based on the Block Least Mean Squares (LMS) algorithm.

`H=dsp.BlockLMSFilter('`*PropertyName*`',`*PropertyValue*`,...)` returns an adaptive FIR filter, `H`, with each specified property set to the specified value.

`H=dsp.BlockLMSFilter(length,blocksize,'`*PropertyName*`',...` *PropertyValue*`,...)` returns an adaptive FIR filter, `H`, with the `Length` property set to `length`, the `BlockSize` property set to `blocksize`, and other specified properties set to the specified values.

Properties   **Length**

Length of FIR filter weights vector

Specify the length of the FIR filter weights vector as a positive integer scalar. The default is `32`.

BlockSize

Number of samples acquired before weight adaptation

Specify the number of samples of the input signal to acquire before the object updates the filter weights. The input frame length must be an integer multiple of the block size. The default is `32`.

StepSizeSource

Source of adaptation step size

Choose to specify the adaptation step size factor as `Property` or `Input port`. The default is `Property`.

StepSize

Adaptation step size

Specify the adaptation step size factor as a scalar, nonnegative numeric value. The default is `0.1`. This property applies only when you set the `StepSizeSource` property to `'Property'`. This property is tunable.

LeakageFactor

Leakage factor used in Leaky LMS algorithm

Specify the leakage factor used in Leaky LMS algorithm as a scalar numeric value between `0` and `1`, both inclusive. When the value is less than `1`, the System object implements a leaky LMS algorithm. The default is `1`, providing no leakage in the adapting algorithm. This property is tunable.

InitialWeights

Initial values of filter weights

Specify the initial values of the filter weights as a scalar or a vector of length equal to the `Length` property value. The default is `0`.

AdaptInputPort

Additional input to enable adaptation of filter weights.

Specify when the object should adapt the filter weights. By default, the value of this property is `false`, and the filter continuously updates the filter weights. When this property is set to `true`, an adaptation control input is provided to the `step` method. If the value of this input is nonzero, the filter continuously updates the filter weights. If the input is zero, the filter weights remain at their current value.

WeightsResetInputPort

Additional input to enable weights reset

Specify whether the FIR filter can reset the filter weights. By default, the value of this property is `false`, and the object does not reset the weights. When this property is set to `true`, a reset control input is provided to the `step` method, and the `WeightsResetCondition` property applies. The object resets the filter weights based on the values of the `WeightsResetCondition` property and the reset input to the `step` method.

WeightsResetCondition

Condition that triggers the resetting of filter weights

Specify the event to reset the filter weights as one of `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. The object resets the filter weights based on the values of this property and the reset input to the `step` method. This property applies only when you set the `WeightsResetInputPort` property to `true`. The default is `Non-zero`.

WeightsOutputPort

Output filter weights

Set this property to `true` to output the adapted filter weights. The default is `true`.

Methods

clone	Create adaptive block LMS filter object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties

dsp.BlockLMSFilter

release	Allow property value and input characteristics changes
reset	Reset internal states of adaptive FIR filter object
step	Filter inputs using Block LMS algorithm

Examples   Remove noise using the Block LMS adaptive algorithm:

```
hblms = dsp.BlockLMSFilter(10, 5);
hblms.StepSize = 0.01;
hblms.WeightsOutputPort = false;
hfilt = dsp.DigitalFilter;
hfilt.TransferFunction = 'FIR (all zeros)';
hfilt.Numerator = fir1(10, [.5, .75]);
x = randn(1000,1); % Noise
d = step(hfilt, x) + sin(0:.05:49.95)'; % Noise + Signal
[y, err] = step(hblms, x, d);
subplot(2,1,1), plot(d), title('Noise + Signal');
subplot(2,1,2), plot(err), title('Signal');
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Block LMS Filter block reference page. The object properties correspond to the block parameters.

See Also   dsp.LMSFilter | dsp.DigitalFilter

Purpose   Create adaptive block LMS filter object with same property values

Syntax

Description   C = clone(H) creates a BlockLMSFilter object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   `N = getNumInputs(H)`

Description   `N = getNumInputs(H)` returns the number of expected inputs, `N`, to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(`*H*`)`.

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step
method.

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

dsp.BlockLMSFilter.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the BlockLMSFilter object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.BlockLMSFilter.reset

Purpose   Reset internal states of adaptive FIR filter object

Syntax   reset(H)

Description reset(H) sets the internal states of the BlockLMSFilter object H to their initial values.

Purpose   Filter inputs using Block LMS algorithm

Syntax
```
[Y,ERR,WTS] = step(H,X,D)
[Y,ERR] = step(H,X,D)
[...] = step(H,X,D,MU)
[...] = step(H,X,D,A)
[...] = step(H,X,D,R)
[Y,ERR,WTS] = step(H,X,D,MU,A,R)
```

Description   [Y,ERR,WTS] = step(H,X,D) filters the input X, using D as the desired
signal, and returns the filtered output in Y. The filter error is in ERR,
and the estimated filter weights is in WTS. The filter weights update
once for every block of data that the object processes.

[Y,ERR] = step(H,X,D) returns only the filtered output in Y and the
filter error in ERR, when the WeightsOutputPort property is false.

[...] = step(H,X,D,MU) uses MU as the step size, when you set the
StepSizeSource property to Input port.

[...] = step(H,X,D,A) uses A as the adaptation control, when you
set the AdaptInputPort property to true. When A is nonzero, the
filter continuously updates the filter weights. When A is zero, the filter
weights remain constant.

[...] = step(H,X,D,R) uses R as a reset signal, when you
set the WeightsResetInputPort property to true. Use the
WeightsResetCondition property to set the reset trigger condition. If
a reset event occurs, the filter resets the filter weights to their initial
values.

[Y,ERR,WTS] = step(H,X,D,MU,A,R) filters input X, using D as the
desired signal, MU as the step size, A as the adaptation control, and R as
the reset signal. The objects returns the filtered output in Y, the filter
error in ERR, and the adapted filter weights in WTS. Set the properties
appropriately to provide all possible inputs.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Buffer input signal

Description   The Buffer object buffers an input signal. The number of samples per
                    channel in the input must equal the difference between the output
                    buffer size and buffer overlap (i.e., Length - OverlapLength). When
                    you set FrameBasedProcessing to true, the number of samples
                    per channel equals the number of rows in the input. If you set
                    FrameBasedProcessing to false, the number of samples per channel
                    equals 1.

                    To buffer an input signal:

                    1 Define and set up your buffer System object. See "Construction" on
                       page 5-165.

                    2 Call step to buffer the input according to the properties of
                       dsp.Buffer. The behavior of step is specific to each object in the
                       toolbox.

Construction   H=dsp.Buffer returns a buffer System object, HBUFF, used to buffer
                    input signals with overlap.

                    H=dsp.Buffer('*PropertyName*',*PropertyValue*,...) returns a buffer
                    object, H, with each specified property set to the specified value.

                    H=dsp.Buffer(LEN,OVRLAP,ICS,'*PropertyName*',...
                    PropertyValue,...) returns a buffer object, H, with Length property
                    set to LEN, OverlapLength property set to OVRLAP, InitialConditions
                    property set to ICS and other specified properties set to the specified
                    values.

Properties   **Length**

                    Number of samples to buffer

                    Specify the number of consecutive samples from each input
                    channel to buffer. You can set this property to any scalar integer
                    greater than 0 of MATLAB built-in numeric data type. The
                    default is 64.

OverlapLength

Amount of overlap between outputs

Specify the number of samples by which consecutive output frames overlap. You can set this property to any scalar integer greater than or equal to `0` of MATLAB built-in numeric data type. The default is `0`.

InitialConditions

Initial output

Specify the value of the object's initial output for cases of nonzero latency as a scalar, vector, or matrix. The default is `0`.

FrameBasedProcessing

Enable frame-based processing

If this property is set to `true`, each column of the input is treated as a separate channel for buffering. If the value of this property is `false`, each element of the input is treated as a separate channel. The default is `true`.

Methods		
	clone	Create instance of an object with the same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes

reset	Reset the internal states of a System object
step	Buffer input signal based on past values

Examples   Create a buffer of 256 samples with 128 sample overlap:

```
hreader=dsp.SignalSource(randn(1024,1),128);
hbuff=dsp.Buffer(256,128);

for i=1:8
    y=step(hbuff,step(hreader));
    % y is of length 256 with 128 samples
    %   from previous input
 end
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Buffer block reference page. The object properties correspond to the block properties, except as noted.

This object processes inputs as frames or as samples by setting the FrameBasedProcessing property. The corresponding block has a temporary **Treat Mx1 and unoriented sample-based signals as** parameter with One channel option for frame behavior and M channels option for sample behavior. See the Buffer block reference page and "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.DelayLine | dsp.Delay

dsp.Buffer.clone

Purpose   Create instance of an object with the same property values

Syntax   C=clone(H)

Description   C=clone(H) creates a `Buffer` System object `C`, with the same property values as `H`.

The `clone` method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.Buffer.getNumOutputs

Purpose   Number of outputs of step method

Syntax   `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, N, of the `step` method

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the Buffer System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.Buffer.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset the internal states of a System object

Syntax   `reset(H)`

Description `reset(H)` resets the internal states of buffer to their initial values.

For many System objects, this method is a no-op. Objects that have internal states will describe in their help what the reset method does for that object.

The reset method is always a no-op for unlocked System objects, as the states may not be allocated when the object is not locked.

dsp.Buffer.step

Purpose   Buffer input signal based on past values

Syntax   Y = step(H,X)

Description   Y = step(H,X) creates output Y based on current input and stored past values of X . Output length equals the Length property.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Estimate of autoregressive (AR) model parameters using Burg method

Description   The BurgAREstimator object computes the estimate of the
                   autoregressive (AR) model parameters using the Burg method.

                   To compute the estimate of the AR model parameters:

                   1 Define and set up your System object. See "Construction" on page
                   5-175.

                   2 Call step to compute the estimate according to the properties of
                   dsp.BurgAREstimator. The behavior of step is specific to each object
                   in the toolbox.

Construction   H = dsp.BurgAREstimator returns a Burg BurgAREstimator System
                   object, H, that performs parametric AR estimation using the Burg
                   maximum entropy method.

                   H = dsp.BurgAREstimator('*PropertyName*',*PropertyValue*,...)
                   returns a Burg AR estimator object, H, with each specified property set
                   to the specified value.

Properties   **AOutputPort**

                   Enable output of polynomial coefficients

                   Set this property to true to output the polynomial coefficients, A,
                   of the AR model the object computes. The default is true. Either
                   the AOutputPort property, the KOutputPort property, or both
                   must be true.

                   KOutputPort

                   Enable output of reflection coefficients

                   Set this property to true to output the reflection coefficients, K,
                   for the AR model that the object computes. The default is false.
                   Either the AOutputPort property, the KOutputPort property, or
                   both must be true.

                   EstimationOrderSource

Source of estimation order

Specify how to determine estimator order as `Auto` or `Property`. When you set this property to `Auto`, the object assumes the estimation order is one less than the length of the input vector. When you set this property to `Property`, the value in EstimationOrder is used. The default is `Auto`.

EstimationOrder

Order of AR model

Set the AR model estimation order to a real positive integer. This property applies when you set the `EstimationOrderSource` to `Property`. The default is `4`.

Methods

clone	Create Burg AR Estimator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Normalized estimate of AR model parameter

Examples

Use the Burg AR Estimator System object to estimate the parameters of an AR model:

```
rng default; % Use default random number generator and seed
noise = randn(100,1); % Normalized white Gaussian noise
x = filter(1,[1 1/2 1/3 1/4 1/5],noise);
hburgarest = dsp.BurgAREstimator(...
```

```
'EstimationOrderSource', 'Property', ...
'EstimationOrder', 4);
[a, g] = step(hburgarest, x);
x_est = filter(g, a, x);
plot(1:100,[x x_est]);
title('Original and estimated signals');
legend('Original', 'Estimated');
```
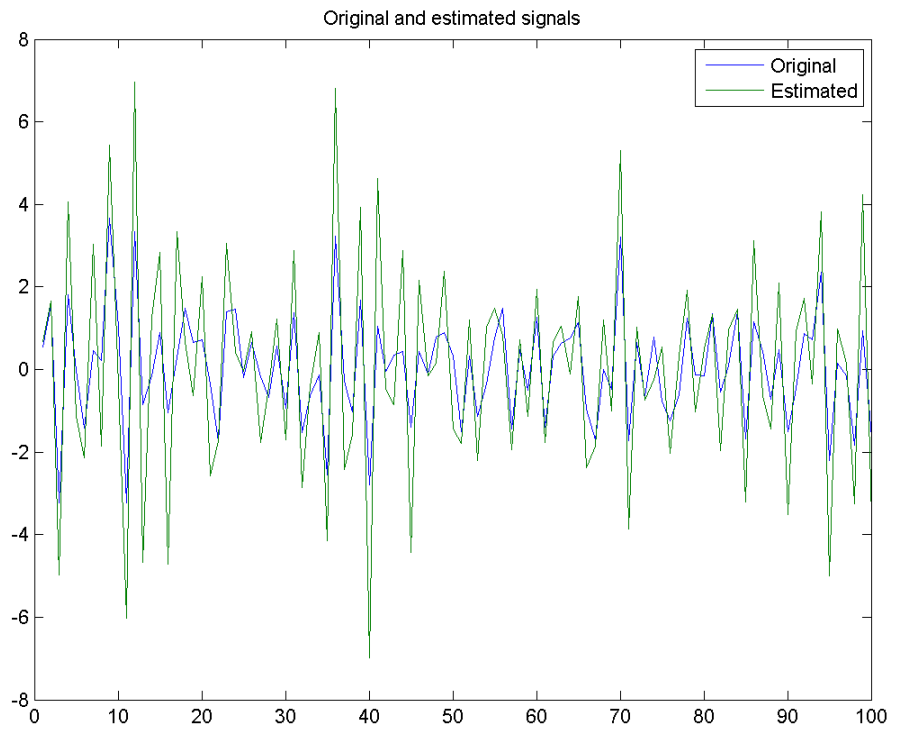

Original and estimated signals

Algorithms   This object implements the algorithm, inputs, and outputs described on the Burg AR Estimator block reference page. The object properties correspond to the block parameters, except:

> **Output(s)** block parameter corresponds to the `AOutputPort` and the `KOutputPort` object properties.

See Also   `dsp.LevinsonSolver`

Purpose   Create Burg AR Estimator object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a BurgAREstimator System object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                 method

                 The getNumOutputs method returns a positive integer that is the
                 number of outputs from the step method. This value will change if you
                 alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the BurgAREstimator System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose　　Allow property value and input characteristics changes

Syntax　　`release(H)`

Description　　`release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note　You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Normalized estimate of AR model parameter

Syntax
```
[A,G] = step(H,X)
[K,G] = step(H,X)
[A,K,G] = step(H,X)
```

Description   `[A,G] = step(H,X)` computes the normalized estimate of the AR model parameters to fit the input, X, in the least square sense. The input X must be a column vector. Output A is a column vector that contains the normalized estimate of the AR model polynomial coefficients in descending powers of z. The scalar G is the AR model gain.

`[K,G] = step(H,X)` returns K, a column vector containing the AR model reflection coefficients when you set the KOutputPort property to true and the AOutputPort property to false.

`[A,K,G] = step(H,X)` returns the AR model polynomial coefficients A, reflection coefficients K, and the scalar gain G when the AOutputPort and KOutputPort properties are both true.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Parametric spectral estimate using Burg method

Description   The BurgSpectrumEstimator object computes a parametric spectral
                   estimate of the input using the Burg method. The object fits an
                   autoregressive (AR) model to the signal by minimizing the forward and
                   backward prediction errors (via least-squares). The AR parameters are
                   constrained to satisfy the Levinson-Durbin recursion.

                   To compute the parametric spectral estimate of the input:

                   1 Define and set up your System object. See "Construction" on page
                   5-185.

                   2 Call step to compute the estimate according to the properties of
                   dsp.BurgSpectrumEstimator. The behavior of step is specific to
                   each object in the toolbox.

Construction   H = dsp.BurgSpectrumEstimator returns an object, H, that estimates
                   the power spectral density (PSD) of the input frame using the Burg
                   method.

                   H =
                   dsp.BurgSpectrumEstimator('*PropertyName*',*PropertyValue*,...)
                   returns a spectrum estimator, H, with each specified property set to
                   the specified value.

Properties   **EstimationOrderSource**

                   Source of estimation order

                   Specify the source of the estimation order as Auto or Property. If
                   you set this property to Auto, the object assumes the estimation
                   order is one less than the length of the input vector. The default
                   value is Property.

                   EstimationOrder

                   Order of AR model

Specify the order of AR model as a real positive integer. This property applies only when you set the `EstimationOrderSource` property to `Property`. The default value is `6`.

FFTLengthSource

Source of FFT length

Specify the source of the FFT length as `Auto` or `Property`. When you set this property to `Auto`, the objects assumes the FFT length is one more than the estimation order. When you set this property to `Property`, the `FFTLength` property value must be an integer power of two.

FFTLength

FFT length as power-of-two integer value

Specify the FFT length as a power-of-two numeric scalar. This property applies when you set the `FFTLengthSource` property to `Property`. The default value is `256`.

SampleRate

Sample rate of input time series

Specify the sampling rate of the original input time series as a positive numeric scalar in hertz. The default value is `1`.

Methods		
	clone	Create estimator object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Estimate of power spectral density

Examples Estimate PSD using the Burg method:

```
x = randn(100,1);
h = dsp.BurgSpectrumEstimator('EstimationOrder', 4);
y = filter(1,[1 1/2 1/3 1/4 1/5],x); % Fourth order AR filter
p = step(h,y); % Uses default FFT length of 256

plot([0:255]/256, p);
title('Burg Method Spectral Density Estimate');
xlabel('Normalized frequency'); ylabel('Power/frequency');
```

Algorithms This object implements the algorithm, inputs, and outputs described on the Burg Method block reference page. The object properties correspond to the block properties.

See Also dsp.BurgAREstimator | dsp.LevinsonSolver

Purpose   Create estimator object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a BurgSpectrumEstimator object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                     the step method.

                     The getNumInputs method returns a positive integer that is the number
                     of expected inputs (not counting the object itself) to the step method.
                     This value will change if you alter any properties that turn inputs on or
                     off. You must call the step method with the number of input arguments
                     equal to the result of getNumInputs(*H*).

dsp.BurgSpectrumEstimator.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the BurgSpectrumEstimator object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.BurgSpectrumEstimator.release

Purpose	Allow property value and input characteristics changes
Syntax	`release(H)`
Description	`release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Estimate of power spectral density

Syntax   Y = step(H,X)

Description   Y = step(H,X) outputs Y, a spectral estimate of input X, using the
                   Burg method.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.CepstralToLPC

Purpose   Convert cepstral coefficients to linear prediction coefficients

Description   The `CepstralToLPC` object converts cepstral coefficients to linear prediction coefficients (LPC).

To convert cepstral coefficients to LPC:

1 Define and set up your System object. See "Construction" on page 5-194.

2 Call `step` to convert the coefficients according to the properties of `dsp.CepstralToLPC`. The behavior of `step` is specific to each object in the toolbox.

Construction   H=dsp.CepstralToLPC returns a System object, H, that converts the cepstral coefficients(CCs) to linear prediction coefficients (LPCs).

H=dsp.CepstralToLPC('*PropertyName*',*PropertyValue*,...) returns a `Cepstral to LPC` object, H, with each specified property set to the specified value.

Properties   **PredictionErrorOutputPort**

Enable prediction error power output

Set this property to `true` to output the prediction error power. The prediction error power is the power of the error output of an FIR analysis filter represented by the LPCs for a given input signal. The default is `false`.

Methods

clone	Create cepstral to LPC object with same property values
getNumInputs	Number of expected inputs to the method
getNumOutputs	Number of outputs of the method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	LPC coefficients from column of cepstral coefficients

Examples   Convert cepstral coefficients to linear prediction coefficients (LPC):

```
cc=[0 0.9920 0.4919 0.3252 0.2418 , ...
0.1917 0.1583 0.1344 0.1165 0.0956]';
hcc2lpc=dsp.CepstralToLPC;
a=step(hcc2lpc,cc);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the LPC to/from Cepstral Coefficients block reference page. The object properties correspond to the block parameters.

See Also   dsp.LPCToCepstral | dsp.LSFToLPC | dsp.RCToLPC

dsp.CepstralToLPC.clone

Purpose   Create cepstral to LPC object with same property values

Syntax   C=clone(H)

Description   C=clone(H) creates a CepstralToLPC System object C, with the same property values as H.

The clone method creates a new unlocked object.

Purpose   Number of expected inputs to the `step` method

Syntax   N = getNumInputs(H)

Description   `N = getNumInputs(H)` returns the number of expected inputs, N, to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

dsp.CepstralToLPC.getNumOutputs

Purpose   Number of outputs of the `step` method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the CepstralToLPC System
object.

The isLocked method returns a logical value that indicates whether
input attributes and nontunable properties for the object are locked. The
object performs an internal initialization the first time the step method
is executed. This initialization locks nontunable properties and input
specifications, such as dimensions, complexity, and data type of the
input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   LPC coefficients from column of cepstral coefficients

Syntax   A=step(H,CC)
                 [A,P]=step(H,CC)

Description A=step(H,CC) computes the linear prediction coefficients (LPC)
                 coefficients, A , from the columns of cepstral coefficients, CC.

                 [A,P]=step(H,CC) converts the columns of the cepstral coefficients
                 CC to the LPCs and returns the prediction error power P when the
                 PredictionErrorOutputPort property is true.

                 Note The object performs an initialization the first time the step
                 method is executed. This initialization locks nontunable properties and
                 input specifications, such as dimensions, complexity, and data type
                 of the input data. If you change a nontunable property or an input
                 specification, the System object issues an error. To change nontunable
                 properties or inputs, you must first call the release method to unlock
                 the object.

dsp.Chirp

Purpose	Generate swept-frequency cosine (chirp) signal

Description

The Chirp object generates a swept-frequency cosine (chirp) signal.

To generate the chirp signal:

1 Define and set up your chirp signal. See "Construction" on page 5-202.

2 Call step to generate the signal according to the properties of dsp.Chirp. The behavior of step is specific to each object in the toolbox.

Construction

H = dsp.Chirp returns a chirp signal, H, with unity amplitude.

H = dsp.Chirp('*PropertyName*',*PropertyValue*,...) returns a chirp signal, H, with each specified property set to the specified value.

Properties

Type

Frequency sweep type

Specify the frequency sweep type as Swept cosine, Linear, Logarithmic, or Quadratic. This property specifies how the output instantaneous frequency sweep varies over time. The default value is Linear.

SweepDirection

Sweep direction

Specify the sweep direction as either Unidirectional or Bidirectional. The default value is Unidirectional.

InitialFrequency

Initial frequency (hertz)

When you set the Type property to Linear, Quadratic, or Logarithmic, this property specifies the initial instantaneous frequency in hertz of the output chirp signal. When you set the Type property to Logarithmic, the value of this property is one

less than the actual initial frequency of the sweep. Also, when the sweep is logarithmic, the initial frequency must be less than the target frequency, specified by the `TargetFrequency` property. This property is tunable. The default value is `1000`.

TargetFrequency

Target frequency (hertz)

When you set the `Type` property to `Linear`, `Quadratic`, or `Logarithmic`, this property specifies the instantaneous frequency of the output signal in hertz at the target time. When you set the `Type` property to `Swept Cosine`, the target frequency is the instantaneous frequency of the output at half the target time. Also, when the sweep is logarithmic, the target frequency must be greater than the initial frequency, specified by the `InitialFrequency` property. This property is tunable. The default value is `4000`.

TargetTime

Target time

When you set the `Type` property to `Linear`, `Quadratic`, or `Logarithmic`, this property specifies the target time in seconds at which the target frequency is reached. When you set the `Type` property to `Swept cosine`, this property specifies the time at which the sweep reaches $2f_{tgt} - f_{init}$ Hz, where f_{tgt} is the `TargetFrequency` and f_{init} is the `InitialFrequency`. The target time should not be greater than the sweep time, specified by the `SweepTime` property. This property is tunable. The default value is `1`.

SweepTime

Sweep time

When you set the `SweepDirection` property to `Unidirectional`, the sweep time in seconds is the period of the output frequency sweep. When you set the `SweepDirection` property to `Bidirectional`, the sweep time is half the period of the output

frequency sweep. The sweep time should be no less than the target time, specified by the `TargetTime`. This property must be a positive numeric scalar and is tunable. The default value is `1`.

InitialPhase

Initial phase

Specify initial phase of the output in radians at time $t = 0$. This property is tunable. The default value is `0`.

SampleRate

Sample rate

Specify the sampling rate of the output in hertz as a positive numeric scalar. The default value is `8000`.

SamplesPerFrame

Samples per output frame

Specify the number of samples to buffer into each output as a positive integer. The default value is `1`.

OutputDataType

Output data type

Specify the output data type as `double` or `single`. The default value is `double`.

Methods		
	clone	Create chirp object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
reset	Reset internal states of chirp object
step	Generate chirp signal

Examples   Generate a bidirectional swept chirp signal:

```
hchirp = dsp.Chirp(...
'SweepDirection', 'Bidirectional', ...
'TargetFrequency', 25, ...
'InitialFrequency', 0,...
'TargetTime', 1, ...
'SweepTime', 1, ...
'SamplesPerFrame', 400, ...
'SampleRate', 400);
plot(step(hchirp));
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Chirp block reference page. The object properties correspond to the block parameters.

See Also   dsp.SineWave

dsp.Chirp.clone

Purpose Create chirp object with same property values

Syntax C = clone(H)

Description C = clone(H) creates a Chirp object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.Chirp.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the Chirp object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset internal states of chirp object

Syntax   `reset(H)`

Description   `reset(H)` sets the internal states of the `Chirp` object `H` to their initial values. After you reset `H`, the frequency sweep restarts from the beginning.

Purpose   Generate chirp signal

Syntax   Y = step(H)

Description   Y = step(H) returns a swept-frequency cosine output, Y.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Decimate input using Cascaded Integrator-Comb filter

Description   The CICDecimator object decimates inputs using a Cascaded
Integrator-Comb filter. Inputs and outputs to the object have signed
fixed-point data types. You must have a Fixed-Point Toolbox license to
use the CICDecimator System object.

To decimate inputs using a CIC filter:

1 Define and set up your System object. See "Construction" on page
5-213.

2 Call step to decimate the input according to the properties of
dsp.CICDecimator. The behavior of step is specific to each object
in the toolbox.

Construction   H = dsp.CICDecimator returns a CICDecimator System object, H, that
you can use to decimate the input with a cascaded integrator-comb
(CIC) decimation filter.

H = dsp.CICDecimator(*'PropertyName'*,*PropertyValue*,...)
returns a CICDecimator object, H, with each specified property set to
the value you specify.

H = dsp.CICDecimator(*R*,*M*,*N*,*'PropertyName'*, *PropertyValue*,
...) returns a CICDecimator object, H, with the DecimationFactor
property set to *R*, the DifferentialDelay property set to *M*, the
NumSections property set to *N*, and any other specified properties set
to the values you specify.

Properties   **DecimationFactor**

Decimation factor of filter

Specify a positive integer amount by which the object decimates
the input. The default is 2.

DifferentialDelay

Differential delay of filter comb sections

Specify a positive integer delay value for the object to use in each comb section of the filter. The default is 1.

NumSections

Number of integrator and comb sections

Specify the number of integrator and comb sections in the CIC filter as a positive integer value. The default is 2.

FixedPointDataType

Fixed-point property setting

Specify the fixed-point data type as one of | Full precision | Minimum section word lengths | Specify word lengths | Specify word and fraction lengths |. The default is Full precision. When you set this property to:

- Full precision — the CICDecimator object automatically determines the word and fraction lengths of the filter sections and output.

- Minimum section word length — the object automatically determines the word and fraction lengths of the filter sections and the fraction length of the output. You must specify the OutputWordLength.

- Specify word lengths — the object automatically determines the fraction lengths of the filter sections and the output. You must specify values for the OutputWordLength and the SectionWordLengths properties.

- Specify word and fraction lengths, — the object does not automatically determine word and fraction lengths for the filter sections or the output. You must specify them using the OutputFractionLength, OutputWordLength, SectionFractionLengths, and SectionWordLengths properties.

SectionWordLengths

Word length of each filter section

Specify the fixed-point word length for each section of the CIC filter as a scalar or a vector of length 2*NumSections. This property applies only when you set the FixedPointDataType property to either Specify word lengths or Specify word and fraction lengths. The default is [16 16 16 16].

SectionFractionLengths

Fraction length of each filter section

Specify the fixed-point fraction length for each section of the CIC filter as a scalar or a vector of length 2*NumSections. This property applies only when you set the FixedPointDataType property to Specify word and fraction lengths. The default is 0.

OutputWordLength

Word length of filter output

Specify the fixed-point word length for the filter output. This property applies only when you set the FixedPointDataType property to Minimum section word lengths, Specify word lengths, orSpecify word and fraction lengths. The default is 32.

OutputFractionLength

Fraction length of filter output

Specify the fixed-point fraction length for the output of the CIC filter. This property applies only when you set the FixedPointDataType property to Specify word and fraction lengths. The default is 0.

Methods

clone	Create CIC Decimator object with same property values
freqz	Frequency response

fvtool	Open filter visualization tool
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
impz	Impulse response
isLocked	Locked status for input attributes and nontunable properties
phasez	Unwrapped phase response
release	Allow property value and input characteristics changes
reset	Reset filter states of CIC decimator object
step	Decimate input using Cascaded Integrator-Comb filter

More "Analysis Methods for Filter System Objects" on page 4-2.

Examples   Decimate signal by a factor of 4 (i.e., downsample the signal from 44.1 kHz to 11.025 kHz):

```
hcicdec = dsp.CICDecimator(4);
%    DecimationFactor = 4
%    default NumSections = 2 and DifferentialDelay = 1
hcicdec.FixedPointDataType = 'Minimum section word lengths';
hcicdec.OutputWordLength = 16;

% Create fixed-point sinusoidal input signal
Fs = 44.1e3;              % Original sampling frequency
n = (0:1023)';            % 1024 samples, 0.0232 sec signal
x = fi(sin(2*pi*1e3/Fs*n),true,16,15);

% Create SignalSource System object
```

```
hsr = dsp.SignalSource(x, 64);

% Decimate output with 16 samples per frame
y = zeros(16,16);
for ii=1:16
     y(ii,:) = step(hcicdec, step( hsr ));
end

% Plot first frame of original and decimated signals.
% Output latency is 2 samples.
gainCIC = ...
   (hcicdec.DecimationFactor*hcicdec.DifferentialDelay)^hcicdec.NumSec
stem(n(1:56)/Fs, double(x(4:59))); hold on;
stem(n(1:14)/(Fs/hcicdec.DecimationFactor),double(y(1,3:end))/gainCIC
xlabel('Time (sec)');ylabel('Signal Amplitude');
legend('Original signal', 'Decimated signal', 'location', 'north');
hold off;
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the CIC Decimation block reference page. The object properties correspond to the block properties, except:

- The **Rate options** block parameter is not supported by the dsp.CICDecimator object.

See Also   dsp.CICInterpolator | dsp.FIRDecimator

dsp.CICDecimator.clone

Purpose Create CIC Decimator object with same property values

Syntax C = clone(H)

Description C = clone(H) creates a CICDecimator System object C, with the same property values as H. The clone method creates a new unlocked object with un-initialized states.

Purpose   Frequency response

Syntax
```
[h,w] = freqz(H)
[h,w] = freqz(H,n)
[h,w] = freqz(H,Name,Value)
freqz(H)
```

Description   [h,w] = freqz(H) returns the complex, 8192–element frequency response vector h, and the corresponding frequencies w in radians/sample.

[h,w] = freqz(H,n) returns the complex, n-element frequency response vector h, and the corresponding frequencies w in radians/sample, using n samples.

[h,w] = freqz(H,Name,Value) returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

freqz(H) uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object H.

dsp.CICDecimator.fvtool

Purpose	Open filter visualization tool
Syntax	`fvtool(H)` `fvtool(H,'Arithmetic',ARITH,...)`
Description	`fvtool(H)` performs an analysis and computes the magnitude response of the filter System object `H`.
	`fvtool(H,'Arithmetic',ARITH,...)` analyzes the filter System object `H`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                 the step method.

                 The getNumInputs method returns a positive integer that is the number
                 of expected inputs (not counting the object itself) to the step method.
                 This value will change if you alter any properties that turn inputs on or
                 off. You must call the step method with the number of input arguments
                 equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

Purpose	Impulse response

Syntax

```
[h,t] = impz(H)
[h,t] = impz(H,Name,Value)
impz(H)
```

Description   `[h,t] = impz(H)` returns the impulse response h, and the corresponding time points t at which the impulse response of H is computed.

`[h,t] = impz(H,Name,Value)` returns the impulse response h, and the corresponding time points t , with additional options specified by one or more Name, Value pair arguments.

`impz(H)` uses FVTool to plot the impulse response of the filter System object H.

Note You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

Purpose   Locked status for input attributes and nontunable properties

Syntax   `isLocked(H)`

Description `isLocked(H)` returns the locked state of the `CICDecimator` System object `H`.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Unwrapped phase response

Syntax   `[phi,w] = phasez(H)`
`[phi,w] = phasez(H,n)`
`[phi,w] = phasez(H,Name,Value)`
`phasez(H)`

Description   `[phi,w] = phasez(H)` returns the 8192–element phase response vector
`phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(H,n)` returns the n-element phase response vector
`phi`, and the corresponding frequencies `w` in radians/sample, using
`n` samples.

`[phi,w] = phasez(H,Name,Value)` returns the phase response and the
corresponding frequencies, with additional options specified by one or
more Name, Value pair arguments.

`phasez(H)` uses FVTool to plot the phase response of the filter System
object `H`.

dsp.CICDecimator.release

Purpose　　Allow property value and input characteristics changes

Syntax　　`release(H)`

Description　　`release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset filter states of CIC decimator object

Syntax   reset(H)

Description   reset(H) resets the filter states of the CICDecimator System object H to zero.

Purpose   Decimate input using Cascaded Integrator-Comb filter

Syntax   Y = step(H,X)

Description Y = step(H,X) decimates the fixed-point input X to produce a
                 fixed-point output Y using the CICDecimator System object H.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

Purpose   Interpolate signal using Cascaded Integrator-Comb filter

Description   The CICInterpolator object interpolates inputs using a Cascaded
                    Integrator-Comb (CIC) filter. Inputs and outputs to the object have
                    signed fixed-point data types. You must have a Fixed-Point Toolbox
                    license to use the CICInterpolator System object.

                    To interpolate inputs using a CIC filter:

                    1 Define and set up your System object. See "Construction" on page
                       5-229.

                    2 Call step to interpolate the input according to the properties of
                       dsp.CICInterpolator. The behavior of step is specific to each object
                       in the toolbox.

Construction   H = dsp.CICInterpolator returns a System object, H, that you can
                    use to interpolate the input with a cascaded integrator-comb (CIC)
                    interpolation filter.

                    H = dsp.CICInterpolator('*PropertyName*',*PropertyValue*,...)
                    returns a CIC interpolation object, H, with each specified property set to
                    the value you specify.

                    H =
                    dsp.CICInterpolator(*R*,*M*,*N*,'*PropertyName*',*PropertyValue*,...)
                    returns a CICInterpolator object, H, with the InterpolationFactor
                    property set to *R*, the DifferentialDelay property set to *M*, the
                    NumSections property set to *N*, and other specified properties set to the
                    values you specify.

Properties   **InterpolationFactor**

                    Interpolation factor of filter

                    Specify a positive integer amount by which the object interpolates
                    the input signal. The default is 2.

                    DifferentialDelay

Differential delay of filter comb sections

Specify a positive integer delay value for the object to use in each comb section of the filter. The default is 1.

NumSections

Number of integrator and comb sections

Specify the number of integrator and comb sections in the CIC filter as a positive integer value. The default is 2.

FixedPointDataType

Fixed-point property setting

Specify the fixed-point data type as one of | Full precision | Minimum section word lengths | Specify word lengths | Specify word and fraction lengths |. The default is Full precision. When you set this property to:

- Full precision — the CICInterpolator object automatically determines the word and fraction lengths of the filter sections and output.

- Minimum section word length — the object automatically determines the word and fraction lengths of the filter sections and the fraction length of the output. You must specify the OutputWordLength.

- Specify word lengths — the object automatically determines the fraction lengths of the filter sections and the output. You must specify values for the OutputWordLength and the SectionWordLengths properties.

- Specify word and fraction lengths — the object does not automatically determine word and fraction lengths for the filter sections or the output. You must specify them using the OutputFractionLength, OutputWordLength, SectionFractionLengths, and SectionWordLengths properties.

SectionWordLengths

Word length of each filter section

Specify the fixed-point word length for each section of the CIC filter as a scalar or a vector of length 2*NumSections. This property applies only when you set the FixedPointDataType property to Specify word lengths or Specify word and fraction lengths. The default is [16 16 16 16].

SectionFractionLengths

Fraction length of each filter section

Specify the fixed-point fraction length for each section of the CIC filter as a scalar or a vector of length 2*NumSections. This property applies only when you set the FixedPointDataType property to Specify word and fraction lengths. The default is 0.

OutputWordLength

Word length of filter output

Specify the fixed-point word length for the filter output. This property applies only when you set the FixedPointDataType property to Minimum section word lengths, Specify word lengths, orSpecify word and fraction lengths. The default is 32.

OutputFractionLength

Fraction length of filter output

Specify the fixed-point fraction length for the output of the CIC filter. This property applies only when you set the FixedPointDataType property to Specify word and fraction lengths. The default is 0.

dsp.CICInterpolator

Methods

clone	Create CIC Interpolator object with same property values
freqz	Frequency response
fvtool	Open filter visualization tool
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
impz	Impulse response
isLocked	Locked status for input attributes and nontunable properties
phasez	Unwrapped phase response
release	Allow property value and input characteristics changes
reset	Reset internal states of CIC Interpolator object
step	Interpolate signal using Cascaded Integrator-Comb filter

More "Analysis Methods for Filter System Objects" on page 4-2.

Examples   Interpolate signal by a factor of 2 (upsample the signal from 22.05 kHz to 44.1 kHz):

```
hcicint = dsp.CICInterpolator(2);
%    default NumSections = 2, DifferentialDelay = 1

% Create fixed-point sinusoidal input signal
Fs = 22.05e3;      % Original sampling frequency
n = (0:511)';      % 512 samples, 0.0113 sec signal
x = fi(sin(2*pi*1e3/Fs*n),true,16,15);
```

```
% Create SignalSource System object
hsr = dsp.SignalSource(x, 32);

% Interpolate output with 64 samples per frame
y = zeros(16,64);
for ii=1:16
    y(ii,:) = step(hcicint,step( hsr ));
end

% Plot first frame of original and interpolated signals.
% Output latency is 2 samples.
gainCIC = ...
   (hcicint.InterpolationFactor*hcicint.DifferentialDelay)...
   ^hcicint.NumSections/hcicint.InterpolationFactor;
stem(n(1:31)/Fs, double(x(1:31)),'r','filled'); hold on;
stem(n(1:61)/(Fs*hcicint.InterpolationFactor), ...
   double(y(1,4:end))/gainCIC,'b');
xlabel('Time (sec)');ylabel('Signal Amplitude');
legend('Original signal', 'Interpolated signal',...
   'location', 'north');
hold off;
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the CIC Interpolation block reference page. The object properties correspond to the block properties, except:

- The **Framing** block parameter is not supported by the dsp.CICInterpolation object. The object always maintains the input frame rate.

- The **Rate options** block parameter is not supported by the dsp.CICInterpolation object.

- The **Input processing** block parameter is not supported by the dsp.CICInterpolation object.

dsp.CICInterpolator

- The object does not have a property that allows you to specify the source of the coefficients. The object cannot import coefficients from an `mfilt` object.

See Also   dsp.CICDecimator | dsp.FIRInterpolator

Purpose　　　　Create CIC Interpolator object with same property values

Syntax　　　　　C = clone(H)

Description　　　C = clone(H) creates a CICInterpolator System object C, with the same property values as H. The clone method creates a new unlocked object with un-initialized states.

dsp.CICInterpolator.freqz

Purpose   Frequency response

Syntax   ```
                 [h,w] = freqz(H)
                 [h,w] = freqz(H,n)
                 [h,w] = freqz(H,Name,Value)
                 freqz(H)
                 ```

**Description**  `[h,w] = freqz(H)` returns the complex, 8192–element frequency
                 response vector `h`, and the corresponding frequencies `w` in
                 radians/sample.

                 `[h,w] = freqz(H,n)` returns the complex, n-element frequency
                 response vector `h`, and the corresponding frequencies `w` in
                 radians/sample, using `n` samples.

                 `[h,w] = freqz(H,Name,Value)` returns the frequency response and
                 the corresponding frequencies, with additional options specified by one
                 or more Name, Value pair arguments.

                 `freqz(H)` uses FVTool to plot the magnitude and unwrapped phase of
                 the frequency response of the filter System object `H`.

**Purpose**    Open filter visualization tool

**Syntax**    fvtool(H)
              fvtool(H,'Arithmetic',ARITH,...)

**Description**    fvtool(H) performs an analysis and computes the magnitude response
                  of the filter System object H.

                  fvtool(H,'Arithmetic',ARITH,...) analyzes the filter System object
                  H, based on the arithmetic specified in the ARITH input. ARITH can
                  be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a
                  double precision filter when the arithmetic input is not specified and
                  the filter System object is in an unlocked state. The 'Arithmetic' input
                  is only relevant for the analysis of filter System objects.

**Purpose**    Number of expected inputs to step method

**Syntax**    N = getNumInputs(H)

**Description**    N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

**Purpose**          Number of outputs of step method

**Syntax**           N = getNumOutputs(H)

**Description**      N = getNumOutputs(H) returns the number of outputs, N, of the step
                    method.

                    The getNumOutputs method returns a positive integer that is the
                    number of outputs from the step method. This value will change if you
                    alter any properties that turn outputs on or off.

| **Purpose** | Impulse response |
|---|---|

**Syntax**

```
[h,t] = impz(H)
[h,t] = impz(H,Name,Value)
impz(H)
```

Description　　`[h,t] = impz(H)` returns the impulse response h, and the corresponding time points t at which the impulse response of H is computed.

`[h,t] = impz(H,Name,Value)` returns the impulse response h, and the corresponding time points t , with additional options specified by one or more Name, Value pair arguments.

`impz(H)` uses FVTool to plot the impulse response of the filter System object H.

Note You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the CICInterpolator System object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.CICInterpolator.phasez

Purpose   Unwrapped phase response

Syntax
```
[phi,w] = phasez(H)
[phi,w] = phasez(H,n)
[phi,w] = phasez(H,Name,Value)
phasez(H)
```

Description   `[phi,w] = phasez(H)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(H,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(H,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(H)` uses FVTool to plot the phase response of the filter System object `H`.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.CICInterpolator.reset

Purpose   Reset internal states of CIC Interpolator object

Syntax   reset(H)

Description   reset(H) resets the internal states of the `CICInterpolator` System
object H to zero.

Purpose   Interpolate signal using Cascaded Integrator-Comb filter

Syntax   Y = step(H,X)

Description   Y = step(H,X) interpolates the fixed-point input X to produce a
fixed-point output Y using the CICInterpolator System object H.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.Convolver

Purpose	Convolution of two inputs
Description	The Convolver computes the convolution of two inputs.

To compute the convolution of two inputs:

1 Define and set up your convolver. See "Construction" on page 5-246.

2 Call step to compute the convolution according to the properties of dsp.Convolver. The behavior of step is specific to each object in the toolbox.

Construction H = dsp.Convolver returns a convolver object, H, that convolves two inputs. For N-D arrays, the convolver computes the convolution column-wise. For arrays, the inputs must have an equal number of columns. If one input is a vector and the other is an N-D array, the convolver computes the convolution of the vector with each column of the N-D array. Convolving inputs of length N and M results in a sequence of length $N+M-1$. Convolving matrices of size M-by-N and P-by-N results in a matrix of size $(M+P-1)$-by-N.

H = dsp.Convolver('*PropertyName*',*PropertyValue*, ...) returns a convolver object, H, with each property set to the specified value.

Properties **Method**

Domain for computing convolutions

Specify the domain in which the convolver performs the convolutions as Time Domain, Frequency Domain, or Fastest. Computing convolutions in the time domain minimizes memory use. Computing convolutions in the frequency domain may require fewer computations depending on the input length. If the value of this property is Fastest, the object computes convolutions in the domain which minimizes the number of computations. The default is Time Domain.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as `Full precision`, `Same as first input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `ProductDataType` property is `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`, `Same as product`, `Same as first input`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `AccumulatorDataType` property is `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as accumulator`, `Same as product`, `Same as first input`, or `Custom`. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `OutputDataType` property is `Custom`. The default is `numerictype([],16,15)`.

Methods

clone	Create convolver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Convolution of inputs

Examples

Convolution of two rectangular sequences:

```
hconv = dsp.Convolver;
x = ones(10,1);
y = step(hconv, x, x);
% Result is a triangular sequence
plot(y);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolution block reference page. The object properties correspond to the block parameters.

See Also

dsp.Autocorrelator | dsp.Crosscorrelator

dsp.Convolver.clone

Purpose   Create convolver object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a convolver object, C, with the same property
                 values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.Convolver.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the convolver object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.Convolver.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Convolution of inputs

Syntax   Y = step(H,A,B)

Description   Y = step(H,A,B) convolves the inputs A and B.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

---

dsp.Counter

Purpose	Count up or down through specified range of numbers

Description

The Counter object counts up or down through a specified range of numbers.

To count up or down through a specified range of numbers:

1 Define and set up your counter. See "Construction" on page 5-256.

2 Call step to count up or down according to the properties of dsp.Counter. The behavior of step is specific to each object in the toolbox.

Construction

H = dsp.Counter returns a counter System object, H, that counts up when the input is nonzero.

H = dsp.Counter('*PropertyName*',*PropertyValue*,...) returns a counter System object, H, with each specified property set to the specified value.

Properties

Direction

Count up or down

Specify the counter direction as Up or Down. The default is Up. This property is tunable.

CountEventInputPort

Add input to specify a count event

Set this property to true to enable a count event input for the internal counter. The internal counter increments or decrements whenever the count event input satisfies the condition you specify in the CountEventCondition property. When you set this property to false, the internal counter is free running, that is, the counter increments or decrements on every call to the step method. The default is true.

CountEventCondition

Condition that increments, decrements, or resets internal counter

Specify the event at the count event input that increments or decrements the counter as `Rising edge`, `Falling edge`, `Either edge` or `Non-zero`.

If you set the `ResetInputPort` and `CountEventInputPort` properties to `true`, the counter is reset when the event you specify for the `CountEventCondition` occurs. This property applies only when you set the `CountEventInputPort` property to `true`.

CounterSizeSource

Source of counter size data type

Specify the source of the counter size data type as `Property` or `Input port`. The default is `Property`.

CounterSize

Range of integer values to count through

Specify the range of integer values to count through before recycling to zero as `8 bits`, `16 bits`, `32 bits` or `Maximum`. The default is `Maximum`.

MaximumCount

Counter's maximum value

Specify the counter's maximum value as a numeric scalar value. This property applies only when you set the `CounterSizeSource` property to `Property` and the `CounterSize` property to `Maximum`. The default is `255`. This property is tunable.

InitialCount

Counter initial value

Specify the initial value for the counter. The default is `0`. This property is tunable.

CountOutputPort

Output count

Set this property to `true` to enable output of the internal count. The default is `true`. You cannot set both `CountOutputPort` and `HitOutputPort` to `false` at the same time.

HitOutputPort

Output hit events

Set this property to `true` to enable output of the hit events. You cannot set both `CountOutputPort` and `HitOutputPort` to `false` at the same time. The default is `true`.

HitValues

Values whose occurrence in count produce a true hit output

Specify an integer scalar or a vector of integers, whose occurrences in the count you want flagged as a hit. This property applies only when you set the `HitOutputPort` property to `true`.

ResetInputPort

Add input to enable internal counter reset

When you set this property to `true`, specify a reset input to the `step` method. When the reset input receives the event you specify for the `CountEventCondition` property, the counter resets. If you set the `CountEventInputPort` property to `false`, the counter resets whenever the reset input is not zero.

SamplesPerFrame

Number of samples in each output frame

Specify the number of samples in each output frame. This property applies only when you set the `CountEventInputPort` property to `false`, indicating a free-running counter. The default is `1`.

CountOutputDataType

Data type of count output

Specify the data type of the count output, CNT, as `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32` or `uint32`. This property applies when you set the `CountOutputPort` property to `true`. The default is `double`.

Methods

clone	Create counter object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of Counter object
step	Increment or decrement the internal counter

Examples   Use Counter System object for counting from 0 to 5:

```
hcounter = dsp.Counter('MaximumCount', 5, ...
'CountOutputPort', true, ...
'HitOutputPort', false, ...
'ResetInputPort', false);
 sgnl = [0 1 0 1 0 1 0 1 0 1 0 1 ];
   for ii = 1:length(sgnl)
 %count at every rising edge of the input signal.
 cnt(ii) = step(hcounter, sgnl(ii));
   end
   cnt
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Counter block reference page. The object properties correspond to the block parameters.

- The `CountEventCondition` object property does not have a free-running option. Set the `CountEventInputPort` property to `false` to obtain the free-running option.

- The `CounterSizeSource` and `CounterSize` object properties correspond to the **Counter size** block parameter.

- The `CountOutputPort` and `HitOutputPort` correspond to the **Output** block parameter.

- There is no object property that corresponds to the **Hit data type** block parameter. The output type is logical in MATLAB. (This logical is different from the popup logical in the block. For the object, logical corresponds to Boolean in the block.)

Purpose   Create counter object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a Counter object C, with the same property
                values as H. The clone method creates a new unlocked object with
                uninitialized states.

dsp.Counter.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, from the
                   step method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

dsp.Counter.isLocked

Purpose　　　　Locked status for input attributes and nontunable properties

Syntax　　　　`isLocked(H)`

Description　　　`isLocked(H)` returns the locked state of the `Counter` System object.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note   You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.Counter.reset

Purpose   Reset internal states of Counter object

Syntax   reset(H)

Description   reset(H) resets the internal states of the counter object H to their
initial values.

Purpose   Increment or decrement the internal counter

Syntax
```
[CNT,HIT] = step(H,EVENT,RESET)
CNT = step(H,EVENT,RESET)
HIT = step(H,EVENT,RESET)
[...] = step(H)
[...] = step(H,EVENT)
```

Description   `[CNT,HIT] = step(H,EVENT,RESET)` increments, decrements, or resets the internal counter as specified by the values of the `EVENT` and `RESET` inputs. The output argument `CNT` denotes the present value of the counter. A trigger event at the `EVENT` input causes the counter to increment or decrement. A trigger event at the `RESET` input resets the counter to its initial state.

`CNT = step(H,EVENT,RESET)` returns the current value of the count when you set the `CountOutputPort` property to `true`, and the `HitOutputPort` property to `false`.

`HIT = step(H,EVENT,RESET)` returns a Boolean value indicating whether the count has reached any of the values specified by the `HitValues` property. This condition applies when you set the `HitOutputPort` property to `true` and the `CountOutputPort` property to `false`.

`[...]  = step(H)` increments or decrements the free-running internal counter when you set the `CountEventInputPort` property to `false` and the `ResetInputPort` property to `false`.

`[...]  = step(H,EVENT)` increments or decrements the internal counter when the `EVENT` input matches the event you specify for the `CountEventCondition` property and you set the `ResetInputPort` property to `false`.

> **Note** The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose   Cross-correlation of two inputs

Description   The `Crosscorrelator` returns the cross-correlation sequence for two discrete-time deterministic inputs. This object can also return the cross-correlation sequence estimate for two discrete-time, jointly wide-sense stationary (WSS), random processes.

To obtain the cross-correlation for two discrete-time deterministic inputs:

1 Define and set up your cross-correlator. See "Construction" on page 5-269.

2 Call `step` to compute the cross-correlation according to the properties of `dsp.Crosscorrelator`. The behavior of `step` is specific to each object in the toolbox.

Construction `H = dsp.Crosscorrelator` returns a cross-correlator object, H, that computes the cross-correlation of two inputs. For N-D arrays, the cross-correlator computes the correlation column-wise. The inputs must have an equal number of columns. If one input is a vector and the other is an N-D array, the cross-correlator computes the cross-correlation of the vector with each column of the N-D array. Cross correlating inputs of length N and M results in a cross-correlation sequence of length $N+M–1$. Cross correlating matrices of size M-by-N and P-by-N results in a matrix of cross-correlation sequences of size $M+P–1$-by-N.

`H = dsp.Crosscorrelator('`*PropertyName*`',`*PropertyValue*`, ...)` returns a cross-correlator, H, with each property set to the specified value.

Properties   **Method**

Domain for computing correlations

Specify the domain for computing correlation as `Time Domain`, `Frequency Domain`, or `Fastest`. Computing correlations in the time domain minimizes memory use. Computing correlations

in the frequency domain may require fewer computations than computing in the time domain depending on the input length. If the value of this property is `Fastest`, the cross-correlator operates in the domain which minimizes the number of computations. The default is `Time Domain`.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of Full Precision, Same as first input, or Custom. The default is Full Precision.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when the ProductDataType property is Custom. The default is numerictype([],32,30).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as Full Precision, Same as product, Same as first input, or Custom. The default is Full Precision.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when the AccumulatorDataType property is Custom. The default is numerictype([],32,30).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of Same as accumulator, Same as product, Same as first input, or Custom. The default is Same as accumulator.

CustomOutputDataType

Output word and fraction lengths

dsp.Crosscorrelator

Specify the output fixed-point type as a scaled `numerictype`
object with a `Signedness` of `Auto`. This property applies only
when the `OutputDataType` property is `Custom`. The default is
`numerictype([],16,15)`.

Methods

clone	Create cross-correlator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Cross-correlation sequence

Definitions

The cross-correlation of two length N deterministic inputs, or
realizations of jointly WSS random processes, x and y, is:

$$r_{xy}(h) = \begin{cases} \sum_{n=0}^{N-h-1} x(n+h)y^*(n) & 0 \le h \le N-1 \\ r_{yx}^*(h) & -(N-1) \le h \le 0 \end{cases}$$

where h is the lag and * denotes the complex conjugate. If the inputs
are realizations of jointly WSS stationary random processes, $r_{xy}(h)$ is an
unnormalized estimate of the theoretical cross-correlation:

$$\rho_{xy}(h) = E\{x(n+h)y^*(n)\}$$

where E{ } is the expectation operator.

Examples   Compute correlation between two signals:

```
hcorr = dsp.Crosscorrelator;
t=[0:0.001:1];
x1=sin(2*pi*2*t)+0.05*sin(2*pi*50*t);
x2=sin(2*pi*2*t);
y=step(hcorr,x1,x2); %computes cross-correlation of x1 and x2
figure,plot(t,x1,'b',t, x2, 'g');
legend('Input signal 1',' Input signal 2')
figure, plot(y); title('Correlated output')
```

Use cross-correlation to detect delay in jointly stationary white
Gaussian noise inputs:

```
S = rng('default');
% white Gaussian noise input
x = randn(100,1);
% Create copy delayed by 10 samples
% x1[n]=x[n-10]
Hdelay = dsp.Delay(10);
x1= step(Hdelay,x);
Hxcorr = dsp.Crosscorrelator;
y = step(Hxcorr,x1,x);
lags = 0:99; %Positive lags
stem(lags,y(100:end),'markerfacecolor',[0 0 1]);
axis([0 99 -125 125]);
xlabel('Lags');
title('Cross-Correlation of Input Noise and Delayed Version');
```

dsp.Crosscorrelator

Cross-Correlation of Input Noise and Delayed Version

The theoretical cross-correlation sequence is identically zero except at lag 10. Note this is not true in the sample cross-correlation sequence, but the estimate demonstrates a peak at the correct lag.

Algorithms   This object implements the algorithm, inputs, and outputs described on the Correlation block reference page. The object properties correspond to the block parameters.

See Also   dsp.Autocorrelator | dsp.Convolver

Purpose   Create cross-correlator object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a cross-correlator object, C, with the same
                property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.Crosscorrelator.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                  method.

                  The getNumOutputs method returns a positive integer that is the
                  number of outputs from the step method. This value will change if you
                  alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the cross–correlator.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

> **Note** You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Cross-correlation sequence

Syntax   Y = step(H,A,B)

Description   Y = step(H,A,B) computes the cross–correlation of A and B and returns
                   the result in Y.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.CumulativeProduct

Purpose   Cumulative product of channel, column, or row elements

Description   The `CumulativeProduct` object computes the cumulative product of channel, column, or row elements.

To compute the cumulative product of channel, column, or row elements:

1 Define and set up your cumulative product object. See "Construction" on page 5-282.

2 Call `step` to compute the cumulative product according to the properties of `dsp.CumulativeProduct`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.CumulativeProduct` returns a cumulative product object, `H`, that computes the cumulative product of input matrix or input vector elements along the default Dimension.

`H = dsp.CumulativeProduct('PropertyName',PropertyValue,...)` returns a cumulative product object, `H`, with each specified property set to the specified value.

Properties   **Dimension**

Computation dimension for cumulative product

Specify the computation dimension as one of | `Channels (running product)` | `Rows` | `Columns` |. The default is `Channels (running product)`.

ResetInputPort

Enable resetting cumulative product via input port

Set this property to `true` to enable resetting the cumulative product. When you set this property to `true`, specify a reset signal to the `step` method to reset the cumulative product. You can access this property when the `Dimension` property is set to `Channels (running product)`. The default is `false`.

ResetCondition

Reset condition for cumulative product

Specify the event on the reset input port that causes resetting the cumulative product to one of | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero` |. This property applies when you set the `ResetInputPort` property to `true` and the `Dimension` property to `Channels (running product)`. The default is `Rising edge`.

FrameBasedProcessing

Enable frame-based processing

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. You can access this property when the `Dimension` property is set to `Channels (running product)`. The default is `false`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest,` | `Round` | `Simplest` | `Zero` |. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`.

IntermediateProductDataType

Intermediate product word and fraction lengths

Specify the intermediate product fixed-point data type as one of | `Same as input` | `Custom` |. The default is `Same as input`.

CustomIntermediateProductDataType

Intermediate product word and fraction lengths

Specify the intermediate product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `IntermediateProductDataType` property to `Custom`. The default is `numerictype([], 16, 15)`.

ProductDataType

Product output word and fraction lengths

Specify the product output fixed-point data type as one of | `Same as input` | `Custom` |. The default is `Same as input`.

CustomProductDataType

Custom product output word and fraction lengths

Specify the product output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numerictype([], 32, 30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | `Same as product output` | `Same as input` | `Custom` |. The default is `Same as input`.

CustomAccumulatorDataType

Custom accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([], 32, 30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | Same as
product output | Same as input | Custom |. The default is
Same as input.

CustomOutputDataType

Custom output word and fraction lengths

Specify the output fixed-point type as a scaled numerictype
object with a Signedness of Auto. This property applies when
you set the OutputDataType property to Custom. The default is
numerictype([], 16, 15).

Methods

clone	Create cumulative product object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset running cumulative product
step	Cumulative product of input along specified dimension for input

Examples   Compute the cumulative product of a matrix:

```
hcprod = dsp.CumulativeProduct;
x = magic(2);
y = step(hcprod,x);
y = step(hcprod,x)
```

dsp.CumulativeProduct

Algorithms   This object implements the algorithm, inputs, and outputs described on the Cumulative Product block reference page. The object properties correspond to the block parameters, except:

> The **Reset port** block parameter corresponds to both the `ResetCondition` and `ResetInputPort` object properties.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the `FrameBasedProcessing` property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.CumulativeSum

Purpose   Create cumulative product object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a CumulativeProduct System object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

dsp.CumulativeProduct.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   `N = getNumInputs(H)` returns the number of expected inputs, `N`, to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                  method.

                  The getNumOutputs method returns a positive integer that is the
                  number of outputs from the step method. This value will change if you
                  alter any properties that turn outputs on or off.

dsp.CumulativeProduct.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the CumulativeProduct object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset running cumulative product

Syntax   reset(H)

Description reset(H) resets the running cumulative product for object H to zero.

Purpose   Cumulative product of input along specified dimension for input

Syntax   Y = step(H,X)
                 Y = step(H,X,R)

Description   Y = step(H,X) computes the cumulative product along the specified
                 dimension for the input X.

                 Y = step(H,X,R) resets the cumulative product object's state based on
                 the ResetCondition property value and the value of the reset signal, R
                 when the ResetInputPort property is true.

                 Note The object performs an initialization the first time the step
                 method is executed. This initialization locks nontunable properties and
                 input specifications, such as dimensions, complexity, and data type
                 of the input data. If you change a nontunable property or an input
                 specification, the System object issues an error. To change nontunable
                 properties or inputs, you must first call the release method to unlock
                 the object.

dsp.CumulativeSum

Purpose　　Cumulative sum of channel, column, or row elements

Description　　The CumulativeSum object computes the cumulative sum of channel, column, or row elements.

To compute the cumulative sum of channel, column, or row elements:

1 Define and set up your cumulative sum object. See "Construction" on page 5-294.

2 Call step to compute the cumulative sum according to the properties of dsp.CumulativeSum. The behavior of step is specific to each object in the toolbox.

Construction　　H = dsp.CumulativeSum returns a cumulative sum System object, H, which computes the running cumulative sum for each channel in the input.

H = dsp.CumulativeSum('*PropertyName*',*PropertyValue*,...) returns a cumulative sum object, H, with each specified property set to the specified value.

Properties　　**Dimension**

Computation dimension for cumulative sum

Specify the computation dimension as one of | Channels (running sum) | Rows | Columns |. The default is Channels (running sum).

ResetInputPort

Enable resetting cumulative sum via input port.

Set this property to true to enable resetting the cumulative sum. When you set this property to true, you also specify a reset input to the step method to reset the cumulative sum. The default is false.

ResetCondition

Reset condition for cumulative sum

Specify the event on the reset input port that resets the cumulative sum as one of | Rising edge | Falling edge | Either edge | Non-zero |. This property applies when you set the ResetInputPort property to true. The default is Rising edge.

FrameBasedProcessing

Enable frame-based processing

Set this property to true to enable frame-based processing. Set this property to false to enable sample-based processing. You can access this property when the Dimension property is Channels (running sum). The default is false.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor, Nearest, Round, Simplest | Zero |. The default is Floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomAccumulatorDataType

Custom accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([], 32, 30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of one of | `Same as accumulator` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Custom output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype([], 16, 15)`.

Methods

clone	Create cumulative sum object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of cumulative sum object to zero
step	Cumulative sum along specified dimension for input

Examples   Use a cumulative sum object to compute the cumulative sum of a matrix:

```
hcsum = dsp.CumulativeSum;
x = magic(2);
y = step(hcsum,x);
y = step(hcsum,x)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Cumulative Sum block reference page. The object properties correspond to the block properties, except:

> The **Reset port** block parameter corresponds to both the ResetCondition and the ResetInputPort object properties

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the FrameBasedProcessing property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.CumulativeProduct

dsp.CumulativeSum.clone

Purpose   Create cumulative sum object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a CumulativeSum object C, with same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

dsp.CumulativeSum.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step
method.

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the CumulativeSum object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.CumulativeSum.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset internal states of cumulative sum object to zero

Syntax   reset(H)

Description reset(H) sets the states for the running cumulative sum object, H, to zero when the Dimension property is set to Channels (running sum).

dsp.CumulativeSum.step

Purpose	Cumulative sum along specified dimension for input
Syntax	`Y = step(H,X)` `Y = step(H,X,R)`
Description	`Y = step(H,X)` computes the cumulative sum along the specified dimension for the input `X`. `Y = step(H,X,R)` resets the System object's state based on the `ResetCondition` property value and the value of the reset signal, `R`. You can only reset the state if the `ResetInputPort` property is `true`.

> **Note** The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose   Discrete cosine transform (DCT)

Description   The DCT object computes the discrete cosine transform (DCT) of input.

To compute the DCT of input:

1 Define and set up your DCT object. See "Construction" on page 5-305.

2 Call step to compute the DCT according to the properties of dsp.DCT. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.DCT returns a discrete cosine transform (DCT) object, H, used to compute the DCT of a real or complex input signal.

H = dsp.DCT('*PropertyName*',*PropertyValue*, ...) returns a DCT object, H, with each property set to the specified value.

Properties   **SineComputation**

Method to compute sines and cosines

Specify how the DCT object computes the trigonometric values as Trigonometric function or Table lookup. This property must be set to Table lookup for fixed-point inputs. The default is Table lookup.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of Ceiling, Convergent, Floor , Nearest, Round, Simplest, or Zero. This property applies when you set the SineComputation property to Table lookup.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. This property applies when you set the `SineComputation` property to `Table lookup`.

SineTableDataType

Sine table word-length designation

Specify the sine table fixed-point data type as one of `Same word length as input` or `Custom`. This property applies when you set the `SineComputation` property to `Table lookup`.

CustomSineTableDataType

Sine table word length

Specify the sine table fixed-point type as an unscaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` property to `Table lookup` and the `SineTableDataType` property to `Custom`. The default is `numerictype([],16)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of `Full precision`, `Same as input`, `Custom`. This property applies when you set the `SineComputation` property to `Table lookup`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` property to `Table lookup` and the `ProductDataType` property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Full precision`, `Same as input`, `Same as product`, or `Custom`. This property applies when you set the `SineComputation` property to `Table lookup`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` property to `Table lookup` and the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of `Full precision`, `Same as input`, `Custom`. This property applies when you set the `SineComputation` property to `Table lookup`. The default is `Full precision`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineComputation` property to `Table lookup` and the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

Methods

clone	Create discrete cosine transform object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Discrete cosine transform (DCT) of input

Examples

Use DCT to analyze the energy content in a sequence:

```
x = (1:128).' + 50*cos((1:128).'*2*pi/40);
hdct = dsp.DCT;
X = step(hdct, x);
% Set the DCT coefficients which represent less
% than 0.1% of the total energy to 0 and
% reconstruct the sequence using IDCT.
[XX, ind] = sort(abs(X),1,'descend');
ii = 1;
while (norm([XX(1:ii);zeros(128-ii,1)]) <= 0.999*norm(XX))
ii = ii+1;
end
disp(['Number of DCT coefficients that represent 99.9%',...
'of the total energy in the sequence: ',num2str(ii)]);
XXt = zeros(128,1);
XXt(ind(1:ii)) = X(ind(1:ii));
hidct = dsp.IDCT;
xt = step(hidct, XXt);
plot(1:128,[x xt]);
legend('Original signal','Reconstructed signal',...
 'location','best');
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the DCT block reference page. The object properties correspond to the block parameters.

See Also

dsp.IDCT | dsp.FFT | dsp.IFFT

Purpose Create discrete cosine transform object with same property values

Syntax C = clone(H)

Description C = clone(H) creates an instance of the current discrete cosine transform object, C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description `N = getNumInputs(H)` returns the number of expected inputs, N, to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

dsp.DCT.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the DCT object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Discrete cosine transform (DCT) of input

Syntax   Y = step(H,X)

Description   Y = step(H,X) computes the DCT of the input X.

> **Note** The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Delay input by specified number of samples or frames

Description   The Delay object delays an input by a specified number of samples or frames.

To delay an input by a specified number of samples or frames:

1 Define and set up your delay object. See "Construction" on page 5-315.

2 Call step to delay the input according to the properties of dsp.Delay. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.Delay returns a delay object, H, to delay the input by one sample.

H = dsp.Delay('*PropertyName*', *PropertyValue*, ...) returns a delay object, H, with each property set to the specified value.

H = dsp.Delay(LEN,'*PropertyName*',*PropertyValue*, ...) returns a delay object, H, with the Length property set to LEN and other specified properties set to the specified values.

Properties   **Units**

Delay units as samples or frames

Specify the delay units as one of | Samples | Frames |. This property applies when you set the FrameBasedProcessing property to true. The default is Samples.

Length

Amount of delay

Specify amount of delay to apply to the input signal. You can set this property to a scalar, a vector, or an array containing nonnegative integers depending on the value of the FrameBasedProcessing property.

If the `FrameBasedProcessing` property is `false`, the value can be a scalar by which to delay all input channels equally, or an N-D array of the same dimensions as the input whose values specify the number of sample intervals to delay each channel of the input.

If the `FrameBasedProcessing` property is `true`, the value can be an integer by which to equally delay all channels or a vector whose length equals the number of input columns (channels).

The default is 1.

InitialConditionsPerChannel

Enable different initial conditions per channel

Set this property to `true` to specify different initial conditions per channel. The default value is `false`.

InitialConditionsPerSample

Enable different initial conditions per sample

Set this property to `true` to specify different initial conditions per sample. The default value is `false`.

InitialConditions

Initial output of delay object

Specify the initial output(s) of the delay object. You can set this property to a scalar, vector, matrix, or a cell array depending on the values of the `FrameBasedProcessing`, `InitialConditionsPerChannel`, `InitialConditionsPerSample`, and `Units` properties. The default is `0`.

If the `FrameBasedProcessing` property is `false`, and the input is an N-D array, the dimensions of this property value have the following requirements:

- If the `InitialConditionsPerChannel` and `InitialConditionsPerSample` properties are both `false`, the property value must be a scalar. If the `InitialConditionsPerChannel` property is `true` and the

InitialConditionsPerSample property is `false`, the
value must have the same dimensions as the input. If the
InitialConditionsPerChannel property is `false` and the
InitialConditionsPerSample property is `true`, the value
must be a vector of length equal to the `Length` property value.

- If the `InitialConditionsPerChannel` and
  `InitialConditionsPerSample` properties are both
  `true`, the property value must be a cell array of the same size
  as the input signal. Each cell of the cell array contains the
  delay values for one channel, and must be a vector of length
  equal to the `Length` property value.

If the `FrameBasedProcessing` property is `true`, and the input is
an M-by-N matrix, the dimensions of this property value must
be as follows:

- If the `InitialConditionsPerChannel` and
  `InitialConditionsPerSample` properties are both
  `false`, the property value must be a scalar.

- If the `InitialConditionsPerChannel` property is `true` and the
  `InitialConditionsPerSample` property is `false`, the value
  must be a vector of length N.

- If the `InitialConditionsPerChannel` property is `false`, the
  `InitialConditionsPerSample` property is `true`, and the `Units`
  property is `Frames`, the value must be a vector of length equal
  to the product of M and the `Length` property value.

- If the `InitialConditionsPerChannel` property is `false`, the
  `InitialConditionsPerSample` property is `true`, and the `Units`
  property is `Samples`, the value must be a vector of length equal
  to the `Length` property value.

- If the `InitialConditionsPerChannel` and
  `InitialConditionsPerChannel` properties are both
  `true`, the property value must be a cell array of size N. Each
  cell of the cell array contains the delay values for one channel.

- If the `Units` property is `Frames`, each cell must be a vector of length equal to the product of *M* and the `Length` property value.

- If the `Units` property is `Samples`, each cell must be a vector of length equal to the `Length` property value.

ResetInputPort

Enable resetting delay states

Specify when the delay object should reset the delay states. By default, the value of this property is `false`, and the object does not reset the delay states. When this property is set to `true`, a reset control input is provided to the `step` method, and the `ResetCondition` property applies. The object resets the delay states based on the values of the `ResetCondition` property and the reset input to the `step` method.

ResetCondition

Reset trigger setting for delay

Specify the event to reset the delay as one of | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero` |. The delay object resets the delay based on the values of this property and the reset input to the `step` method. This property applies when you set the `ResetInputPort` property to `true`. The default is `Non-zero`.

FrameBasedProcessing

Enable frame-based processing

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. The default is `true`.

Methods	clone	Create delay object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	reset	Reset delay states
	step	Apply delay to input

Examples   Delay input by five samples:

```
hdelay1 = dsp.Delay(5);
% Output is [0 0 0 0 0 1 2 3 4 5]'
y = step(hdelay1, (1:10)');
```

Delay input by one frame:

```
hdelay2 = dsp.Delay;
hdelay2.Units = 'Frames';
hdelay2.Length = 1;

% Output is zeros(10,1)
y1 = step(hdelay2, (1:10)');

% Output is (1:10)'
y2 = step(hdelay2, (11:20)');
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Delay block reference page. The object properties correspond to the block parameters.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the `FrameBasedProcessing` property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.VariableIntegerDelay | dsp.VariableFractionalDelay.

Purpose   Create delay object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a delay object, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

dsp.Delay.getNumInputs

Purpose　　　Number of expected inputs to step method

Syntax　　　`N = getNumInputs(H)`

Description　`N = getNumInputs(H)` returns the number of expected inputs, N, to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

dsp.Delay.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the delay object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.Delay.reset

Purpose   Reset delay states

Syntax   reset(H)

Description reset(H) resets the states of the delay object H to the values specified
in the `InitialConditions` property.

After you invoke the `step` method for a nonzero input, the delay object
states may change and invoking the `step` method again without
invoking the `reset` method may produce different outputs for identical
inputs.

For example:

```
H = dsp.Delay(5);
% Delay input by 5 samples
y = step(H, (1:10)');
% Output is [0 0 0 0 0 1 2 3 4 5]'
% Invoke step without reset
y1 = step(H,(1:10)');
% Output is [6 7 8 9 10 1 2 3 4 5]'
% Reset states
reset(H);
y2 = step(H,(1:10)');
% Output is [0 0 0 0 0 1 2 3 4 5]'
```

Purpose   Apply delay to input

Syntax   Y = step(H,X)
Y = step(H,X,R)

Description   Y = step(H,X) adds delay to input X to return output Y.

Y = step(H,X,R) adds delay to X , and selectively resets the delay
object's state based on the value of reset input R and the value of the
ResetCondition property. You can use this option only when you set
the ResetInputPort property to true.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.DelayLine

Purpose	Rebuffer sequence of inputs with one-sample shift

Description   The `DelayLine` object rebuffers a sequence of inputs with one-sample shift.

To rebuffer a sequence of inputs with one-sample shift:

1 Define and set up your delay line object. See "Construction" on page 5-328.

2 Call `step` to rebuffer the sequence of inputs according to the properties of `dsp.DelayLine`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.DelayLine` returns a delay line System object, `H`, that buffers the input samples into a sequence of overlapping or underlapping matrix outputs.

`H = dsp.DelayLine('`*PropertyName*`',`*PropertyValue*`,...)` returns a delay line System object, `H`, with each specified property set to the specified value.

`H = dsp.DelayLine(DELAYSIZE,INITIAL,'`*PropertyName*`',`*PropertyValue*`,...)` returns a delay line System object, `H`, with the `Length` property set to `DELAYSIZE`, `InitialConditions` property set to `INITIAL` and other specified properties set to the specified values.

Properties   **Length**

Number of rows in output matrix

Specify the number of rows in the output matrix as a scalar positive integer. The default is `64`.

InitialConditions

Initial delay line output

Set the value of the object's initial output as one of | scalar | vector | matrix |.

For vector outputs, the following selections apply for the `InitialConditions` property:

- A vector of the same size
- A scalar value that you want repeated across all elements of the initial output

For matrix outputs, the following selections apply for the `InitialConditions` property:

- A matrix of the same size
- A vector (equal to the length of the number of matrix rows) that repeats across all columns of the initial output
- A scalar that repeats across all elements of the initial output

The default is `0`.

DirectFeedthrough

Enable passing input data to output without extra frame delay

When you set this property to `true`, there is no input data delay by an extra frame before it is available at the output buffer. Instead, the input data is available immediately at the output. When you set this property to `false`, there is one frame delay on the output. The default is `false`.

EnableOutputInputPort

Enable selective output linearization

The object internally uses a circular buffer, even though the output is linear. To obtain a valid output, the object must linearize the circular buffer. When this property is `true`, the object uses an additional Boolean input to determine if a valid output calculation is needed. If Boolean the input value is `1`, the object's output is linearized and thus valid. If Boolean the input value is `0`, the output is not linearized and is invalid. This allows the object to be

more efficient when each step does not require the tapped delay line output. When you set this property to `false`, the output is always linearized and valid. The default is `false`.

HoldPreviousValue

Hold previous valid value for invalid output

If you set this property to `true`, the most recent, valid value is held on the output. If you set this property to `false`, the signal on the output is invalid data. This property applies only when you set the `EnableOutputInputPort` property to `true`. The default is `false`.

Methods

clone	Create delay line object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of delay line object
step	Delayed version of input

Examples

Use a delay line object with a delay line size of 4 samples:

```
hdelayline = dsp.DelayLine( ...
   'Length', 4, ...
   'DirectFeedthrough', true, ...
   'InitialConditions', -2, ...
   'EnableOutputInputPort', true, ...
   'HoldPreviousValue', true);
```

```
en = logical([1 1 0 1 0]);
y = zeros(4,5);
for ii = 1:5
    y(:,ii) = step(hdelayline, ii, en(ii));
end
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Delay Line block reference page. The object properties correspond to the block properties, except as noted.

This object processes inputs as separate channels (frames). The corresponding block has a temporary **Treat Mx1 and unoriented sample-based signals as** parameter with a One channel option for frame-based behavior and an M channels option for sample-based behavior. See the Delay Line block reference page and "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.Delay

dsp.DelayLine.clone

Purpose   Create delay line object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a DelayLine object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.DelayLine.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, from the
                  step method

                  The getNumOutputs method returns a positive integer that is the
                  number of outputs from the step method. This value will change if you
                  alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the DelayLine object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.DelayLine.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

> **Note** You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset internal states of delay line object

Syntax   `reset(H)`

Description   `reset(H)` sets the internal delay buffers of the `DelayLine` object `H` to
                  their initial conditions.

Purpose   Delayed version of input

Syntax   Y = step(H,X)
                 Y = step(H,X,EN)

Description   Y = step(H,X) returns the delayed version of input X. Y is an output
                 matrix with the same number of rows as the delay line size. Each
                 column of X is treated as a separate channel.

                 The System object rebuffers a sequence of Mi-by-N matrix inputs into a
                 sequence of Mo-by-N matrix outputs, where Mo is the output frame size
                 specified by the Length property. Depending on whether Mo is greater
                 than, less than, or equal to the input frame size, Mi, the output frames
                 can be underlapped or overlapped. Each of the N input channels is
                 rebuffered independently:

                 • When $Mo > Mi$, the output frame overlap is the difference between
                   the output and input frame size, Mo–Mi.

                 • When $Mo < Mi$, the output is underlapped; the object discards the
                   first Mi–Mo samples of each input frame so that only the last Mo
                   samples are buffered into the corresponding output frame.

                 • When $Mo = Mi$, the output data is identical to the input data, but is
                   delayed by the latency of the object.

                 Y = step(H,X,EN) selectively outputs the delayed version of input
                 X depending on the Boolean input EN. This occurs only when you set
                 the EnableOutputInputPort property to true. If EN is false, use the
                 HoldPreviousValue property to specify if the object should hold the
                 previous output value(s).

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose   Digitally downconvert input signal

Description   The `DigitalDownConverter` object digitally downconverts the input signal.

To digitally downconvert the input signal:

1 Define and set up your digital down converter. See "Construction" on page 5-340.

2 Call `step` to downconvert the input according to the properties of `dsp.DigitalDownConverter`. The behavior of `step` is specific to each object in the toolbox.

Construction   `h = dsp.DigitalDownConverter` returns a digital down-converter (DDC) System object, `h`. The object frequency downconverts the input signal by multiplying it with a complex exponential with center frequency equal to the value in the `CenterFrequency` property. The object downsamples the frequency down-converted signal using a cascade of three decimation filters. When you set the `FilterSpecification` property to `Design parameters`, the DDC object designs the decimation filters according to the filter parameters that you set in the filter-related object properties. In this case the filter cascade consists of a CIC decimator, a CIC compensator and a third FIR decimation stage. The following block diagram shows the architecture of the digital down converter.

The scaling section normalizes the CIC gain and the oscillator power. It may also contain a correction factor to achieve the desired ripple specification. When you set the **Oscillator** property to InputPort, the normalization factor does not include the oscillator power factor. Depending on the setting of the **DecimationFactor** property, you may be able to bypass the third filter stage. When the arithmetic is double or single precision, the CIC interpolator is implemented as a simple FIR interpolator with a boxcar response. A true CIC interpolator can only operate with fixed-point signals. The CIC filter is emulated with an FIR filter so that you can run simulations with floating-point data.

The following block diagram represents the DDC arithmetic with single or double-precision, floating-point inputs.

The following block diagram represents the DDC arithmetic with signed fixed-point inputs.

- `WL` is the word length of the input, and `FL` is the fraction length of the input.

- The input of each filter is cast to the data type specified in the **FiltersInputDataType** and **CustomFiltersInputDataType** properties.

- The oscillator output is cast to a word length equal to the input word length plus one. The fraction length is equal to the input word length minus one.

- The scaling at the output of the CIC decimator consists of coarse- and fine-gain adjustments. The coarse gain is achieved using the `reinterpretcast` function on the CIC decimator output. The fine gain is achieved using full-precision multiplication.

The following figure depicts the coarse- and fine-gain operations.

If the normalization gain is G, (where 0<G≤1), then:

- `WLcic` is the word length of the CIC decimator output and `FLcic` is the fraction length of the CIC decimator output

- `F1 = abs(nextpow2(G))`, indicating the part of G achieved using bit shifts (coarse gain)

- `F2` = fraction length specified by the **FiltersInputDataType** and **CustomFiltersInputDataType** properties

- `fg = fi((2^F1)*G, true, 16)` , indicating that the remaining gain cannot be achieved with a bit shift (fine gain)

`h = dsp.DigitalDownConverter(Name,Value)` returns a DDC object, h, with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as (`Name1,Value1,...,NameN,ValueN`).

Properties

SampleRate

Sample rate of input signal

Set this property to a positive scalar value, greater than or equal to twice the value of the `CenterFrequency` property. The default is 30 Megahertz.

DecimationFactor

Decimation factor

Set this property to a positive, integer scalar, or to a 1-by-2 or 1-by-3 vector of positive integers.

When you set this property to a scalar the object automatically chooses the decimation factors for each of the three filtering stages. The scalar value you specify cannot be a prime number.

When you set this property to a 1-by-2 vector, the object bypasses the third filter stage and sets the decimation factor of the first and second filtering stages to the values in the first and second vector elements respectively. When you set the `FilterSpecification` property to `Design parameters`, both elements of the `DecimationFactor` vector must be greater than one. When you set the `FilterSpecification` property to `Coefficients`, the first element of the `DecimationFactor` vector must be greater than 1.

When you set this property to a 1-by-3 vector, the i th element of the vector specifies the decimation factor for the i th filtering stage. When you set the `FilterSpecification` property to `Design parameters`, the first and second elements of the `DecimationFactor` vector must be greater than one and the third element must be 1 or 2. When you set the `FilterSpecification` property to `Coefficients`, the first element of the `DecimationFactor` vector must be greater than 1. When you set the `FilterSpecification` property to `Coefficients`, you must set the `DecimationFactor` property to a 1-by-3 or 1-by-2 vector. The default is 100.

FilterSpecification

Filter specification

Set the filter specification as one of `Design parameters` | `Coefficients`. The default is `Design parameters`. The DDC object performs decimation using a cascade of three decimation filters. The first filter stage is always a CIC decimator. When you set the `FilterSpecification` property to `Design parameters`, the object designs the cascade of decimation filters internally, according to a set of parameters that you specify using the filter-related object properties. In this case, the second and third stages of the cascade consist of a CIC compensator, and a halfband or lowpass FIR decimator respectively. When you set the `FilterSpecification` property to `Coefficients`, you specify an arbitrary set of filter coefficients for the second and third filter stages using the `SecondFilterCoefficients`, and `ThirdFilterCoefficients` properties respectively. You specify the number of CIC sections using the `NumCICSections` property. In all cases, the third filter stage can be bypassed by setting the `DecimationFactor` property appropriately.

When the input data type is double or single the object implements an *N*-section CIC decimation filter as an FIR filter with a response that corresponds to a cascade of *N* boxcar filters. A true CIC filter with actual comb and integrator sections is implemented when the input data is of a fixed point type.

MinimumOrderDesign

Minimum order filter design

When you set this property to true the object designs filters with the minimum order that meets the passband ripple, stopband attenuation, passband frequency, and stopband frequency specifications that you set using the `PassbandRipple`, `StopbandAttenuation`, `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties. When you set this property to false, the object designs filters with orders that you specify in the `NumCICSections`, `SecondFilterOrder`, and `ThirdFilterOrder` properties. The filter designs meet the passband and stopband frequency specifications that you set using the Bandwidth, `StopbandFrequencySource`, and `StopbandFrequency` properties. This property applies when you set the `FilterSpecification` property to `Design parameters`. The default is true.

NumCICSections

Number of sections of CIC decimator

Set this property to a positive, integer scalar. This property applies when you set the `FilterSpecification` property to `Design parameters` and the `MinimumOrderDesign` property to false, or when you set the `FilterSpecification` property to `Coefficients`. The default is 3.

SecondFilterCoefficients

Coefficients of second filter stage

Set this property to a double precision row vector of real coefficients that correspond to an FIR filter. Usually, the response of this filter should be that of a CIC compensator since a CIC decimation filter precedes the second filter stage. This property applies when you set the `FilterSpecification` property to `Coefficients`. The default is 1.

ThirdFilterCoefficients

Coefficients of third filter stage

Set this property to a double precision row vector of real coefficients that correspond to an FIR filter. When you set the `DecimationFactor` property to a 1-by-2 vector, the object ignores the value of the `ThirdFilterCoefficients` property because the third filter stage is bypassed. This property applies when you set the `FilterSpecification` property to `Coefficients`. The default is `1`.

SecondFilterOrder

Order of CIC compensation filter stage

Set this property to a positive, integer scalar. This property applies when you set the `FilterSpecification` property to `Design parameters` and the `MinimumOrderDesign` property to false. The default is `12`.

ThirdFilterOrder

Order of third filter stage

Set this property to a positive, integer, even scalar. When you set the `DecimationFactor` property to a 1-by-2 vector, the object ignores the `ThirdFilterOrder` property because the third filter stage is bypassed. This property applies when you set the `FilterSpecification` property to `Design parameters` and the `MinimumOrderDesign` property to false. The default is `10`.

Bandwidth

Two sided bandwidth of input signal in Hertz

Set this property to a positive, integer scalar. The object sets the passband frequency of the cascade of filters to one-half of the value that you specify in the `Bandwidth` property. This property applies when you set the `FilterSpecification` property to `Design parameters`. The default is 200 kilohertz.

StopbandFrequencySource

Source of stopband frequency

Specify the source of the stopband frequency as one of `Auto` | `Property`. The default is `Auto`. When you set this property to `Auto`, the object places the cutoff frequency of the cascade filter response at approximately F_c = `SampleRate`/M/2 Hertz, where M is the total decimation factor that you specify in the `DecimationFactor` property. The object computes the stopband frequency as $F_{stop} = F_c + TW/2$. TW is the transition bandwidth of the cascade response computed as $2*(F_c$-$F_p)$, and the passband frequency, F_p, equals `Bandwidth`/2. This property applies when you set the `FilterSpecification` property to `Design parameters`.

StopbandFrequency

Stopband frequency in Hertz

Set this property to a double precision positive scalar. This property applies when you set the `FilterSpecification` property to `Design parameters` and the `StopbandFrequencySource` property to `Property`. The default is 150 kilohertz.

PassbandRipple

Passband ripple of cascade response in decibels.

Set this property to a double precision, positive scalar. When you set the `MinimumOrderDesign` property to true, the object designs the filters so that the cascade response meets the passband ripple that you specify in the `PassbandRipple` property. This property applies when you set the `FilterSpecification` property to `Design parameters` and the `MinimumOrderDesign` property to true. The default is `0.1` decibels.

StopbandAttenuation

Stopband attenuation of cascade response in decibels

Set this property to a double precision, positive scalar. When you set the `MinimumOrderDesign` property to true, the object designs the filters so that the cascade response meets the stopband attenuation that you specify in the

StopbandAttenuation property. This property applies when you set the FilterSpecification property to Design parameters and the MinimumOrderDesign property to true. The default is 60 decibels.

Oscillator

Type of oscillator

Specify the oscillator as one of Sine wave | NCO | Input port. The default is Sine wave. When you set this property to Sine wave, the object frequency down converts the input signal using a complex exponential obtained from samples of a sinusoidal trigonometric function. When you set this property to NCO the object performs frequency down conversion with a complex exponential obtained using a numerically controlled oscillator (NCO). When you set this property to Input port, the object performs frequency down conversion using the complex signal that you set as an input to the step method.

CenterFrequency

Center frequency of input signal in Hertz

Specify this property as a double precision positive scalar that is less than or equal to half the value of the SampleRate property. The object down converts the input signal from the passband center frequency you specify in the CenterFrequency property, to 0 Hertz. This property applies when you set the Oscillator property to Sine wave or NCO. The default is 14 Megahertz.

NumAccumulatorBits

Number of NCO accumulator bits

Specify this property as an integer scalar in the range [1 128]. This property applies when you set the Oscillator property to NCO. The default is 16.

NumQuantizedAccumulatorBits

Number of NCO quantized accumulator bits

Specify this property as an integer scalar in the range [1 128]. The value you specify in this property must be less than the value you specify in the NumAccumulatorBits property. This property applies when you set the Oscillator property to NCO. The default is 12.

Dither

Dither control for NCO

When you set this property to true, a number of dither bits specified in the NumDitherBits property will be used to apply dither to the NCO signal. This property applies when you set the Oscillator property to NCO. The default is true.

NumDitherBits

Number of NCO dither bits

Specify this property as an integer scalar smaller than the number of accumulator bits that you specify in the NumAccumulatorBits property. This property applies when you set the Oscillator property to NCO and the Dither property to true. The default is 4.

Fixed-Point Properties

SecondFilterCoefficientsDataType

Data type of second filter coefficients

Specify the second filter coefficients data type as Same as input | Custom. The default is Same as input. This property applies when you set the FilterSpecification property to Coefficients.

CustomSecondFilterCoefficientsDataType

Fixed-point data type of second filter coefficients

Specify the second filter coefficients fixed-point type as a scaled numerictype object with a Signedness of Auto. This property

applies when you set the SecondFilterCoefficientsDataType property to Custom. The default is numerictype([],16,15).

ThirdFilterCoefficientsDataType

Data type of third filter coefficients

Specify the third filter coefficients data type as Same as input |Custom. The default is Same as input. This property applies when you set the FilterSpecification property to Coefficients.

CustomThirdFilterCoefficientsDataType

Fixed-point data type of third filter coefficients

Specify the third filter coefficients fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the ThirdFilterCoefficientsDataType property to Custom. The default is numerictype([],16,15).

FiltersInputDataType

Data type of input of each filter stage

Specify the data type at the input of the first, second, and third (if it has not been bypassed) filter stages as one of Same as input | Custom. The default is Same as input. The object casts the data at the input of each filter stage according to the value you set in this property.

CustomFiltersInputDataType

Fixed-point data type of input of each filter stage

Specify the filters input fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the FiltersInputDataType property to Custom. The default is numerictype([],16,15).

OutputDataType

Data type of output

dsp.DigitalDownConverter

Specify the data type of output as `Same as input` | `Custom`. The default is `Same as input`.

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a scaled numerictype object with a Signedness of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

Methods		
clone	Create digital down converter object with same property values
fvtool	Visualize response of filter cascade
getDecimationFactors	Get decimation factors of each filter stage
getFilterOrders	Get orders of decimation filters
getFilters	Get handles to decimation filter objects
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
groupDelay	Group delay of filter cascade
isLocked	Locked status for input attributes and non-tunable properties
release	Allow property value and input characteristics changes
step	Digitally down convert input signal
visualizeFilterStages	Display response of filter stages

Copy Semantics   Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples   Create a digital up converter object that up samples a 1 KHz sinusoidal signal by a factor of 20 and up converts it to 50 KHz. Create a digital down converter object that down converts the signal to 0 Hz and down samples it by a factor of 20.

```
% Create a sine wave generator to obtain the 1 KHz
% sinusoidal signal with a sample rate of 6 KHz.
Fs = 6e3; % Sample rate
hSig = dsp.SineWave('Frequency',1000,'SampleRate',...
Fs,'SamplesPerFrame',1024);
x = step(hSig); % generate signal

% Create a DUC object. Use minimum order filter designs and set
% passband ripple to 0.2 dB and the stopband attenuation to 55 dB.
% Set the double sided signal bandwidth to 2 KHz.
hDUC = dsp.DigitalUpConverter(...
 'InterpolationFactor', 20,...
 'SampleRate', Fs,...
 'Bandwidth', 2e3,...
 'StopbandAttenuation', 55,...
 'PassbandRipple',0.2,...
 'CenterFrequency',50e3);

% Create a DDC object. Use minimum order filter designs and set the
% passband ripple to 0.2 dB and the stopband attenuation to 55 dB.
hDDC = dsp.DigitalDownConverter(...
  'DecimationFactor',20,...
  'SampleRate', Fs*20,...
  'Bandwidth', 3e3,...
  'StopbandAttenuation', 55,...
  'PassbandRipple',0.2,...
  'CenterFrequency',50e3);

% Create a spectrum estimator to visualize the signal spectrum before
```

```
% up converting, after up converting, and after down converting.
hSEst = spectrum.welch; hSEst.SegmentLength = length(x)/10;
hOpts=psdopts(hSEst);
set(hOpts, 'Fs', Fs, 'SpectrumType','twosided','CenterDC',true);
figure; psd(hSEst,x,hOpts)
title('Spectrum of baseband signal x')

% Up convert the signal and visualize the spectrum
xUp = step(hDUC,x); % up convert
hOpts.Fs = Fs*20; hSEst.SegmentLength = length(xUp)/10;
figure; psd(hSEst,xUp,hOpts)
title('Spectrum of up converted signal xUp')

% Down convert the signal and visualize the spectrum
xDown = step(hDDC,xUp); % down convert
hOpts.Fs = Fs; hSEst.SegmentLength = length(xDown)/10;
figure; psd(hSEst,xDown,hOpts)
title('Spectrum of down converted signal xDown')

% Visualize the response of the decimation filters
visualizeFilterStages(hDDC)
```

For additional examples using this System object, see the following
demos:

- Digital Up and Down Conversion for Family Radio Service
- Design and Analysis of a Digital Down Converter

See Also   dsp.DigitalUpConverter

Purpose   Create digital down converter object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a DigitalDownConverter System object, C, with
                   the same property values as H.

                   The clone method creates a new unlocked object with uninitialized
                   states.

Purpose   Visualize response of filter cascade

Syntax
```
fvtool(H)
fvtool(H,..,'Arithmetic',ARITH,...)
fvtool(H,..., PROP1, VALUE1,PROP2,VALUE2,...)
```

Description   fvtool(H) plots the magnitude response of the cascade of filters. By default, the object plots the cascade response up to the second CIC null frequency (or to the first when only one CIC null exists). When you set the `FilterSpecification` property to `Design parameters` the method plots a mask based on the filter specifications.

fvtool(H,..,'Arithmetic',ARITH,...) specifies the arithmetic of the filter cascade. You set input ARITH to `double`, `single`, or `fixed-point`. When object H is in an unlocked state you must specify the arithmetic. When object H is in a locked state the arithmetic input is ignored.

fvtool(H,..., PROP1, VALUE1,PROP2,VALUE2,...) launches FVTool and sets the specified FVTool properties to the specified values.

Purpose   Get decimation factors of each filter stage

Syntax   M = getDecimationFactors(H)

Description M = getDecimationFactors(H)) returns a vector, M, with the
                 decimation factors of each filter stage. If the third filter stage is
                 bypassed, then M is a 1-by-2 vector containing the decimation factors
                 of the first and second filter stages in the first and second elements
                 respectively. If the third filter stage is not bypassed then M is a 1-by-3
                 vector containing the decimation factors of the first, second and third
                 filter stages.

dsp.DigitalDownConverter.getFilters

Purpose   Get handles to decimation filter objects

Syntax   S = getFilters(H)
                 getFilters(H,'Arithmetic',ARITH)

Description   S = getFilters(H) returns a structure, S, with copies of the filter
                 System objects and the CIC normalization factor that form the
                 decimation filter cascade. The ThirdFilterStage structure field is
                 empty if the third filter stage has been bypassed. The CIC normalization
                 factor equals the inverse of the CIC filter gain. In some cases, this gain
                 includes a correction factor to ensure that the cascade response meets
                 the ripple specifications.

                 getFilters(H,'Arithmetic',ARITH) specifies the arithmetic of the
                 filter stages. You can set ARITH to double, single, or fixed-point.
                 When object H is in an unlocked state, you must specify the arithmetic
                 input. When object H is in a locked state, the arithmetic input is ignored.

                 When H is in an unlocked state, and you specify the arithmetic as
                 fixed-point, the getFilters method returns filter System objects.
                 The custom coefficient data type properties of these System objects
                 are set to the values that the dsp.DigitalDownConverter System
                 object uses to process data when you call the step method. All other
                 fixed-point properties are set to their default values.

                 When H is in a locked state, and the input to the step method is of a
                 fixed-point data type, the getFilters method returns filter System
                 objects. All fixed-point properties of these System objects are set to
                 the exact values that the dsp.DigitalDownConverter System object
                 uses to process the data.

Purpose　　Get orders of decimation filters

Syntax　　`S = getFilterOrders(H)`

Description　　`S = getFilterOrders(H)` returns a structure, `S`, that contains the orders of the interpolation filter stages. The `ThirdFilterOrder` structure field will be empty if the third filter stage has been bypassed.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                       method

                       The getNumOutputs method returns a positive integer that is the
                       number of outputs from the step method. This value will change if you
                       alter any properties that turn outputs on or off.

dsp.DigitalDownConverter.groupDelay

Purpose   Group delay of filter cascade

Syntax   D = groupDelay(H,N)
                 [D,F] = groupDelay(H,N)

Description D = groupDelay(H,N) returns a vector of group delays, D, evaluated
                 at N frequency points equally spaced around the upper half of the unit
                 circle. If you don't specify N, it defaults to 8192.

                 [D,F] = groupDelay(H,N) returns a vector of frequencies, F, at which
                 the group delay has been computed.

Purpose   Locked status for input attributes and non-tunable properties

Syntax   Y = isLocked(H)

Description   Y = isLocked(H) returns the locked state, Y, of the
                   DigitalUpConverter object.

                   The isLocked method returns a logical value that indicates whether
                   input attributes and nontunable properties for the object are locked. The
                   object performs an internal initialization the first time the step method
                   is executed. This initialization locks nontunable properties and input
                   specifications, such as dimensions, complexity, and data type of the
                   input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections. This method lets you change any properties or input characteristics.

Purpose   Digitally down convert input signal

Syntax   Y = step(H,X)
                Y = step(H,X,Z)

Description   Y = step(H,X) takes a real or complex input column vector X and outputs a frequency down converted and down sampled signal Y. The length of input X must be a multiple of the decimation factor. X can be of data type double, single, signed integer, or signed fixed point (fi objects). The length of Y is equal to the length of X divided by the DecimationFactor. When the data type of X is double or single precision, the data type of Y is the same as that of X. When the data type of X is of a fixed point type, the data type of Y is defined by the OutputDataType property.

Y = step(H,X,Z) uses the complex input, Z, as the oscillator signal used to frequency down convert input X, when you set the Oscillator property to Input port. The length of Z must be equal to the length of X. Z can be of data type double, single, signed integer, or signed fixed point (fi objects).

dsp.DigitalDownConverter.visualizeFilterStages

Purpose   Display response of filter stages

Syntax
```
visualizeFilterStages(H)
visualizeFilterStages(H,'Arithmetic',ARITH)
hfvt = visualizeFilterStages(H)
```

Description   `visualizeFilterStages(H)` plots the magnitude response of the filter stages and of the cascade response. When you set the `FilterSpecification` property to `Design parameters` the method plots a mask based on the filter specifications. By default, the object plots the response of the filters up to the second CIC null frequency (or to the first when only one CIC null exists).

`visualizeFilterStages(H,'Arithmetic',ARITH)` specifies the arithmetic of the filter stages. You set input `ARITH` to `double`, `single`, or `fixed-point`. When object `H` is in an unlocked state you must specify the arithmetic. When object `H` is in a locked state the arithmetic input is ignored.

`hfvt = visualizeFilterStages(H)` returns the handle to the FVTool object.

Purpose Static or time-varying digital filter

Description The DigitalFilter object filters each channel of the input using static or time-varying digital filter implementations.

To filter each channel of the input using digital filter implementation:

1 Define and set up your IIR digital filter. See "Construction" on page 5-367.

2 Call step to filter each channel according to the properties of dsp.DigitalFilter. The behavior of step is specific to each object in the toolbox.

Construction H = dsp.DigitalFilter returns a default IIR digital filter object, H, which independently filters each channel of the input over successive calls to the step method, using a specified digital filter implementation. The default numerator coefficients are [1 2] and denominator coefficients are [1 0.1].

H = dsp.DigitalFilter('*PropertyName*',*PropertyValue*, ...) returns a digital filter object, H, with each property set to the specified value.

Properties **TransferFunction**

Type of filter transfer function

Specify the type of digital filter transfer function as one of | IIR (poles & zeros) | IIR (all poles) | FIR (all zeros) |. The default is IIR (poles & zeros).

Structure

Filter structure

Specify the filter structure.

When you set the TransferFunction property to FIR (all zeros), you can specify the filter structure as one of |

Direct form | Direct form symmetric | Direct form antisymmetric | Direct form transposed | Lattice MA |. The default is Direct form.

When you set the TransferFunction property to IIR (all poles), you can specify the filter structure as Direct form, Direct form transposed, or Lattice AR. The default is Direct form.

When you set the TransferFunction to IIR (poles & zeros), you can specify the filter structure as Direct form I, Direct form I transposed, Direct form II, Direct form II transposed, Biquad direct form I (SOS), Biquad direct form I transposed (SOS), Biquad direct form II (SOS), or Biquad direct form II transposed (SOS). The biquad filter structure does not apply when you set the CoefficientsSource property to Input port. The default is Direct form II transposed.

CoefficientsSource

Source of filter coefficients

Specify the source of the filter coefficients as one of | Property | Input port |. The default is Property. When you specify Input port, the digital filter object updates the time-varying filter once every frame, when the FrameBasedProcessing property is true. When the FrameBasedProcessing property is false, it updates once every sample.

Numerator

Numerator coefficients

Specify the filter numerator coefficients as a real or complex numeric vector. This property applies when you set the TransferFunction property to FIR (all zeros), the CoefficientsSource property to Property, and the Structure property is not set to Lattice MA. This property also applies when you set the TransferFunction property to IIR (poles & zeros), the CoefficientsSource to Property, and the

TransferFunction property to `Direct form`, `Direct form symmetric`, `Direct form antisymmetric`, or `Direct form transposed`. The default is `[1 2]`. This property is tunable.

Denominator

Denominator coefficients

Specify the filter denominator coefficients as a real or complex numeric vector. This property applies when you set the `TransferFunction` property to `IIR (all poles)`, the `CoefficientsSource` property to `Property` and the `Structure` property is not set to `Lattice AR`. This property also applies when you set the `TransferFunction` property to `IIR (poles & zeros)`, the `CoefficientsSource` to `Property`, and the `TransferFunction` property to `Direct form`, `Direct form symmetric`, `Direct form antisymmetric`, or `Direct form transposed`. When the `TransferFunction` property is `IIR (poles & zeros)`, the numerator and denominator must have the same complexity. The default is `[1 0.1]`. This property is tunable.

ReflectionCoefficients

Reflection coefficients of lattice filter structure

Specify the reflection coefficients of a lattice filter as a real or complex numeric vector. This property applies when you set the `TransferFunction` property to `FIR (all zeros)`, the `Structure` property to `Lattice MA`, and the `CoefficientsSource` property to `Property`. This property also applies when you set the `TransferFunction` property to `IIR (all poles)`, the `Structure` property to `Lattice AR`, and the `CoefficientsSource` property to `Property`. The default is `[0.2 0.4]`. This property is tunable.

SOSMatrix

SOS matrix of biquad filter structure

Specify the second-order section (SOS) matrix of a biquad filter as an M-by-6 matrix, where M is the number of sections in the

filter. Each row of the SOS matrix contains the numerator and denominator coefficients of the corresponding filter section. The first three elements of each row are the numerator coefficients and the last three elements are the denominator coefficients. You can use real or complex coefficients. This property applies when you set the `TransferFunction` property to `IIR (poles and zeros)`, the `CoefficientsSource` property to `Property`, and the `Structure` property to `Biquad direct form I (SOS)`, `Biquad direct form I transposed (SOS)`, `Biquad direct form II (SOS)`, or `Biquad direct form II transposed (SOS)`. The default is [1 0.3 0.4 1 0.1 0.2]. This property is tunable.

ScaleValues

Scale values of biquad filter structure

Specify the scale values to apply before and after each section of a biquad filter. `ScaleValues` must be a scalar or a vector of length M+1, where M is the number of sections. If you specify a scalar value, that value is used as the gain value before the first section of the second-order filter and the rest of the gain values are set to 1. If you specify a vector of M+1 values, each value is used for a separate section of the filter. This property applies when you set the `TransferFunction` property to `IIR (poles & zeros)`, the `CoefficientsSource` property to `Property`, and the `Structure` property to `Biquad direct form I (SOS)`, `Biquad direct form I transposed (SOS)`, `Biquad direct form II (SOS)`, or `Biquad direct form II transposed (SOS)`. The default is 1. This property is tunable.

IgnoreFirstDenominatorCoefficient

Assume first denominator coefficient is 1

Setting this Boolean property to `true` reduces the number of computations to produce the digital filter output by omitting the first denominator term, $a0$, (poles side) in the filter structure. The object output is invalid when you set this property to `true` when the first denominator coefficient is not always 1 for your time-varying filter. The object ignores this property for

fixed-point inputs, because this object does not support nonunity *a0* coefficients for fixed-point inputs.

This property applies when you set the `CoefficientsSource` property to `Input port`, the `TransferFunction` property to `IIR (all poles)`, and the `Structure` property is not set to `Lattice AR`. This property also applies when you set the `CoefficientsSource` property to `Input port`, the `TransferFunction` property to `IIR (poles & zeros)`, and the `Structure` property to `Direct form`, `Direct form symmetric`, `Direct form antisymmetric`, or `Direct form transposed`. The default is `true`.

InitialConditions

Initial conditions for all poles or all zeros filter

Specify the initial conditions of the filter states. When the `TransferFunction` property is `FIR (all zeros)` or `IIR (all poles)`, the number of states or delay elements equals the number of reflection coefficients for the lattice structure, or the number of filter coefficients–1 for the other direct form structures. When the `TransferFunction` property is `IIR (poles & zeros)`, the number of states for direct form II and direct form II transposed structures equals the `max(N,M)`–1, where `N` and `M` are the number of poles and zeros, respectively.

You can specify the initial conditions as a scalar, vector, or matrix. If you specify a scalar value, the digital filter object initializes all delay elements in the filter to that value. If you specify a vector whose length equals the number of delay elements in the filter, each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector of initial conditions to each channel of the input signal.

If you specify a vector whose length equals the product of the number of input channels and the number of delay elements in the filter, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel.

If you specify a matrix with the same number of rows as the number of delay elements in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. This property applies when you do not set the `Structure` property to `Direct Form I`, `Direct Form I transposed`, `Biquad direct form I (SOS)`, or `Biquad direct form I transposed (SOS)`. The default is 0.

NumeratorInitialConditions

Initial conditions on zeros side

Specify the initial conditions of the filter states on the side of the filter structure with the zeros. This property applies when you set the `TransferFunction` property to `IIR (poles & zeros)` and the `Structure` property is `Direct form I`, `Direct form I transposed`, `Biquad direct form I (SOS)`, or `Biquad direct form I transposed (SOS)`. The default is `0`. See the `InitialConditions` property for information on setting initial conditions.

DenominatorInitialConditions

Initial conditions on poles side

Specify the initial conditions of the filter states on the side of the direct form I (noncanonic) filter structure with the poles. This property applies when you set the `TransferFunction` property to `IIR (poles & zeros)` and the `Structure` property to `Direct form I`, `Direct form I transposed`, `Biquad direct form I (SOS)`, or `Biquad direct form I transposed (SOS)`. The default is 0. See the `InitialConditions` property for information on setting initial conditions.

FrameBasedProcessing

Enable frame-based processing

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. The default is `true`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`.

TapSumDataType

Tap sum word and fraction lengths

Specify the tap sum fixed-point data type as one of | `Same as input` | `Custom` |. This property applies when you set the `TransferFunction` property to `FIR (all zeros)` and the `Structure` property to either `Direct form symmetric` or `Direct form antisymmetric`. The default is `Same as input`.

CustomTapSumDataType

Custom tap sum word and fraction lengths

Specify the tap sum fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `TapSumDataType` property to `Custom`. The default is `numerictype([],32,30)`.

MultiplicandDataType

Multiplicand word and fraction lengths

Specify the multiplicand fixed-point data type as one of | `Same as output` | `Custom` |. This property applies when you set the `Structure` property to either `Direct form I transposed` or `Biquad direct form I transposed (SOS)`. The default is `Same as output`.

CustomMultiplicandDataType

Custom multiplicand word and fraction lengths

Specify the multiplicand fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `MultiplicandDataType` property to `Custom`. The default to `numerictype([],32,30)`.

SectionInputDataType

Section input word and fraction lengths

Specify the section input fixed-point data type as one of | `Same as input` | `Custom` |. Setting this property also sets the `SectionOutputDataType` property to the same value. This property applies when you set the `TransferFunction` property to `IIR (poles & zeros)` and the `Structure` property to one of the biquad filter structures. The default is `Same as input`.

CustomSectionInputDataType

Custom section input word and fraction lengths

Specify the section input fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SectionInputDataType` property to `Custom`. The `CustomSectionInputDataType` and `CustomSectionOutputDataType` property values must have equal word lengths. The default is `numerictype([],16,15)`.

SectionOutputDataType

Section output word and fraction lengths

Specify the section output fixed-point data type as one of | `Same as input` | `Custom` |. Setting this property also sets

the `SectionInputDataType` property to the same value. This property applies when you set the `TransferFunction` property to `IIR (poles & zeros)` and the `Structure` property to one of the biquad filter structures. The default is `Same as input`.

CustomSectionOutputDataType

Custom section output word and fraction lengths

Specify the section output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SectionOutputDataType` property to `Custom`. The `CustomSectionInputDataType` and `CustomSectionOutputDataType` property values must have equal word lengths. The default is `numerictype([],16,15)`.

NumeratorCoefficientsDataType

Numerator coefficients word and fraction lengths

Specify the numerator coefficients fixed-point data type as one of | `Same word length as input` | `Custom` |. Setting this property also sets the `DenominatorCoefficientsDataType` and the `ScaleValuesDataType` properties, if applicable, to the same value. This property applies when you set the `CoefficientsSource` property to `Property` and the `TransferFunction` property is not set to `IIR (all poles)`. The default is `Same word length as input`.

CustomNumeratorCoefficientsDataType

Custom numerator coefficients word and fraction lengths

Specify the numerator coefficients fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `NumeratorCoefficientsDataType` property to `Custom`. The `CustomNumeratorCoefficientsDataType`, `CustomDenominatorCoefficientsDataType` and `ScaleValuesDataType` properties, if applicable, must have equal word lengths. The default is `numerictype([],16,15)`.

DenominatorCoefficientsDataType

Denominator coefficients word and fraction lengths

Specify the denominator coefficients fixed-point data type as one of | Same word length as input | Custom |. Setting this property also sets the NumeratorCoefficientsDataType and ScaleValuesDataType properties, if applicable, to the same value. This property applies when you set the CoefficientsSource property to Property and the TransferFunction property is not set to FIR (all zeros). The default is Same word length as input.

CustomDenominatorCoefficientsDataType

Custom denominator coefficients word and fraction lengths

Specify the denominator coefficients fixed-point type as a numerictype object with a Signedness of Auto. This property applies when you set the DenominatorCoefficientsDataType property to Custom. The CustomNumeratorCoefficientsDataType, CustomDenominatorCoefficientsDataType and CustomScaleValuesDataType properties, if they apply, must have equal word lengths. The default is numerictype([],16,15).

ScaleValuesDataType

Scale values word and fraction lengths

Specify the scale values fixed-point data type as one of | Same word length as input | Custom |. Setting this property also sets the NumeratorCoefficientsDataType and the DenominatorCoefficientsDataType properties, if they apply, to the same value. This property applies when you set the CoefficientsSource property to Property, the TransferFunction property to IIR (poles & zeros), and the Structure property to one of the biquad filter structures. The default is Same word length as input.

CustomScaleValuesDataType

Custom scale values word and fraction lengths

Specify the scale values fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `CoefficientsSource` property to `Property` and the `ScaleValuesDataType` property to `Custom`. The `CustomNumeratorCoefficientsDataType`, `CustomDenominatorCoefficientsDataType`, and the `CustomScaleValuesDataType` properties, if applicable, must have equal word lengths. The default is `numerictype([],16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | `Same as input` | `Custom` |. The default is `Same as input`.

CustomProductDataType

Custom product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type to one of | `Same as input` | `Same as product` | `Custom` |. The default is `Same as product`.

CustomAccumulatorDataType

Custom accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

StateDataType

State word and fraction lengths

Specify the state fixed-point data type as one of | Same as input | Same as accumulator | Custom |. This property does not apply to any of the direct form or direct form I filter structures. The default is Same as accumulator.

CustomStateDataType

Custom state word and fraction lengths

Specify the state fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the StateDataType property to Custom. The default is numerictype([],16,15).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | Same as input | Same as accumulator | Custom |. The default is Same as accumulator.

CustomOutputDataType

Custom output word and fraction lengths

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the OutputDataType property to Custom. The default is numerictype([],16,15).

Methods

clone	Create static or time-varying digital filter with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of digital filter
step	Filter input with digital filter object

Examples

Use an FIR filter to apply a low pass filter to a waveform with two sinusoidal components:

```
t = [0:63]./32e3;
xin = (sin(2*pi*4e3*t)+sin(2*pi*12e3*t)) / 2;

hSR = dsp.SignalSource(xin', 4);
hLog = dsp.SignalSink;
hFilt = dsp.DigitalFilter;
hFilt.TransferFunction = 'FIR (all zeros)';
hFilt.Numerator = fir1(10,0.5);

while ~isDone(hSR)
    input = step(hSR);
    filteredOutput = step(hFilt,input);
    step(hLog,filteredOutput);
end

filteredResult = hLog.Buffer;
sper = spectrum.periodogram;
psd(sper,filteredResult,'Fs',32e3)
```

dsp.DigitalFilter

Algorithms   This object implements the algorithm, inputs, and outputs described on the Digital Filter block reference page. The object properties correspond to the block parameters.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the `FrameBasedProcessing` property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.BiquadFilter

Purpose   Create static or time-varying digital filter with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a digital filter object, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

dsp.DigitalFilter.getNumInputs

Purpose	Number of expected inputs to step method
Syntax	N = getNumInputs(H)
Description	N = getNumInputs(H) returns the number of expected inputs, N, to the step method.
	The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the digital filter.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.DigitalFilter.reset

Purpose   Reset internal states of digital filter

Syntax   reset(H)

Description reset(H) resets the filter states of the digital filter object,H, to their
initial values of 0. The initial filter state values correspond to the initial
conditions for the difference equation defining the filter. After the step
method applies the digital filter object to nonzero input data, the states
may be nonzero. Invoking the step method again without first invoking
thereset method may produce different outputs for an identical input.

For example:

```
H = dsp.DigitalFilter;
H.TransferFunction = 'FIR (all zeros)';
H.Numerator = fir1(20,0.25);
n = 0:100;
x = cos(0.2*pi*n)+sin(0.8*pi*n);
y = step(H,x);
% Filter states are nonzero
% Invoke step method again without resetting states
y1 = step(H,x);
isequal(y,y1) % returns 0
% Now reset filter states to 0
reset(H)
% Invoke step method
y2 = step(H,x);
isequal(y,y2) % returns a 1
```

Purpose　　Filter input with digital filter object

Syntax　　　`Y = step(HFILT,X)`
　　　　　　　`Y = step(HFILT,X,COEFF)`
　　　　　　　`Y = step(HFILT,X,NUM,DEN)`

Description　`Y = step(HFILT,X)` filters the real or complex input signal X using the digital filter, H, to produce the output Y . The digital filter object operates on each channel of the input signal independently over successive calls to step method.

`Y = step(HFILT,X,COEFF)` uses the time-varying numerator or denominator coefficients, COEFF , to filter the input signal X and produce the output Y . You can use this option when you set the `TransferFunction` property to either FIR (all zeros) or IIR (all poles) and the `CoefficientsSource` property to Input port.

`Y = step(HFILT,X,NUM,DEN)` uses the time-varying numerator coefficients NUM and the time-varying denominator coefficients DEN to filter the input signal X and produce the output Y . You can use this option when you set the `TransferFunction` property to IIR (poles and zeros) and the `CoefficientsSource` property to Input port.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose	Digitally upconvert input signal

Description   The `DigitalUpConverter` object digitally upconverts the input signal.

To digitally up convert the input signal:

1 Define and set up your digital up converter. See "Construction" on page 5-388.

2 Call `step` to upconvert the input according to the properties of `dsp.DigitalUpConverter`. The behavior of `step` is specific to each object in the toolbox.

Construction   `h = dsp.DigitalUpConverter` returns a digital up-converter (DDC) System object, `h`. The object up samples the input signal using a cascade of three interpolation filters. This object frequency upconverts the up sampled signal by multiplying it with a complex exponential with center frequency equal to the value in the `CenterFrequency` property. When you set the `FilterSpecification` property to 'Design parameters', the DUC object designs the interpolation filters according to the filter specifications that you set in the filter-related object properties. In this case the filter cascade consists of a first FIR interpolation stage, a CIC compensator, and a CIC interpolator. The following block diagram shows the architecture of the digital up converter.

The scaling section normalizes the CIC gain and the oscillator power. It may also contain a correction factor to achieve the desired ripple specification. Depending on the setting of the **InterpolationFactor** property, you may be able to bypass the first filter stage. When the arithmetic is double or single precision, the CIC interpolator is implemented as a simple FIR interpolator with a boxcar response. A true CIC interpolator can only operate with fixed-point signals. The CIC filter is emulated with an FIR filter so that you can run simulations with floating-point data.

The following diagram represents the DUC arithmetic with single or double-precision, floating-point inputs.

The following block diagram represents the DUC arithmetic with signed fixed-point inputs.

- `WL` is the word length of the input, and `FL` is the fraction length of the input.

- The output of each filter is cast to the data type specified in the **FiltersOutputDataType** and **CustomFiltersOutputDataType** properties. The casting of the CIC output occurs after the scaling factor is applied.

- The oscillator output is cast to a word length equal to the **FiltersOutputDataType** word length plus one. The fraction length is equal to the **FiltersOutputDataType** word length minus one.

- The scaling at the output of the CIC interpolator consists of coarse- and fine-gain adjustments. The coarse gain is achieved using the `reinterpretcast` function on the CIC interpolator output. The fine gain is achieved using full-precision multiplication.

The following figure depicts the coarse- and fine-gain operations.

If the normalization gain is G, (where $0 < G \leq 1$), then:

- WLcic is the word length of the CIC interpolator output and FLcic is the fraction length of the CIC interpolator output

- F1 = abs(nextpow2(G)), indicating the part of G achieved using bit shifts (coarse gain)

- F2 = fraction length specified by the **FiltersOutputDataType** and **CustomFiltersOutputDataType** properties

- fg = fi((2^F1)*G, true, 16), indicating that the remaining gain cannot be achieved with a bit shift (fine gain)

h = dsp.DigitalUpConverter('*PropertyName*','*PropertyValue*') returns a DUC object, h, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

SampleRate

Sample rate of input signal

Set this property to a positive scalar. The value of this property multiplied by the total interpolation factor must be greater than

or equal to twice the value of the `CenterFrequency` property. The default is 30e6.

InterpolationFactor

Set this property to a positive, integer scalar, or to a 1-by-2 or 1-by-3 vector of positive integers. When you set this property to a scalar the object automatically chooses the interpolation factors for each of the three filtering stages. The scalar value you specify cannot be a prime number. When you set this property to a 1-by-2 vector, the object bypasses the first filter stage and sets the interpolation factor of the second and third filtering stages to the values in the first and second vector elements respectively. When you set the `FilterSpecification` property to `Design parameters`, both elements of the `InterpolationFactor` vector must be greater than one. When you set the `FilterSpecification` property to `Coefficients`, the second element of the `InterpolationFactor` vector must be greater than 1. When you set this property to a 1-by-3 vector, the ith element of the vector specifies the interpolation factor for the ith filtering stage. When you set the `FilterSpecification` property to `Design parameters`, the second and third elements of the `InterpolationFactor` vector must be greater than one and the first element must be 1 or 2. When you set the `FilterSpecification` property to `Coefficients`, the third element of the `InterpolationFactor` vector must be greater than 1. When you set the `InterpolationFactor` property to `Coefficients`, you must set the `InterpolationFactor` property to a 1-by-3 or 1-by-2 vector . The default is 100.

FilterSpecification

Set the filter specification as one of `Design parameters` |`Coefficients`. The default is `Design parameters`. The DUC object performs interpolation using a cascade of three interpolation filters. The third filter stage is always a CIC interpolator. When you set this property to `Design parameters`, the object designs the cascade of interpolation filters internally, according to a set of parameters that you specify using the

filter-related object properties. In this case, the first and second stages of the cascade consist of a halfband or lowpass FIR interpolator, and a CIC compensator respectively. When you set this property to `Coefficients`, you specify an arbitrary set of filter coefficients for the first and second stages using the `FirstFilterCoefficients`, and `SecondFilterCoefficients` properties respectively. You specify the number of CIC sections using the `NumCICSections` property. In all cases, the first filter stage can be bypassed by setting the `InterpolationFactor` property appropriately.

When the input data type is double or single the object implements an N-section CIC interpolation filter as an FIR filter with a response that corresponds to a cascade of *N* boxcar filters. A true CIC filter with actual comb and integrator sections is implemented when the input data is of a fixed point type.

MinimumOrderDesign

Minimum order filter design.

When you set this property to true the object designs filters with the minimum order that meets the passband ripple, stopband attenuation, passband frequency, and stopband frequency specifications that you set using the `PassbandRipple`, `StopbandAttenuation`, `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties. When you set this property to false, the object designs filters with orders that you specify in the `FirstFilterOrder`, `SecondFilterOrder`, and `NumCICSections` properties. The filter designs meet the passband and stopband frequency specifications that you set using the `Bandwidth`, `StopbandFrequencySource`, and `StopbandFrequency` properties. This property applies when you set the `FilterSpecification` property to `Design parameters`. The default is true.

FirstFilterCoefficients

Coefficients of first filter stage.

Set this property to a double precision row vector of real coefficients that correspond to an FIR filter. When you set the `InterpolationFactor` property to a 1-by-2 vector, the object ignores the value of the `FirstFilterCoefficients` property because the first filter stage is bypassed. This property applies when you set the `FilterSpecification` property to `Coefficients`. The default is 1.

SecondFilterOrder

Order of CIC compensation filter stage

Set this property to a positive, integer scalar. This property applies when you set the `FilterSpecification` property to `Design parameters` and the `MinimumOrderDesign` property to false. The default is 12.

SecondFilterCoefficients

Coefficients of second filter stage.

Set this property to a double precision row vector of real coefficients that correspond to an FIR filter. Usually, the response of this filter should be that of a CIC compensator since a CIC interpolation filter follows the second filter stage. This property applies when you set the `FilterSpecification` property to `Coefficients`. The default is 1.

FirstFilterOrder

Order of first filter stage

Set this property to a positive, integer, even scalar. When you set the `InterpolationFactor` property to a 1-by-2 vector, the object ignores the `FirstFilterOrder` property because the first filter stage is bypassed. This property applies when you set the `FilterSpecification` property to `Design parameters` and the `MinimumOrderDesign` property to false. The default is 10.

NumCICSections

Number of sections of CIC interpolator

Set this property to a positive, integer scalar. This property applies when you set the FilterSpecification property to Design parameters and the MinimumOrderDesign property to false, or when you set the FilterSpecification property to Coefficients. The default is 3.

Bandwidth

Two sided bandwidth of input signal in Hertz .

Set this property to a positive, integer scalar. The object sets the passband frequency of the cascade of filters to one-half of the value that you specify in the Bandwidth property. This property applies when you set the FilterSpecification property to Design parameters. The default is 200e3 Hertz.

StopbandFrequencySource

Source of stopband frequency.

Specify the source of the stopband frequency as one of Auto | Property. The default is Auto. When you set this property to Auto, the object places the cutoff frequency of the cascade filter response at approximately F_c = SampleRate/2 Hertz, and computes the stopband frequency as $F_{stop} = F_c + TW/2$. TW is the transition bandwidth of the cascade response, computed as $2*(F_c\text{-}F_p)$, and the passband frequency, F_p, equals Bandwidth/2. This property applies when you set the FilterSpecification property to Design parameters.

StopbandFrequency

Stopband frequency in Hertz

Set this property to a double precision positive scalar. This property applies when you set the FilterSpecification property to Design parameters and the StopbandFrequencySource property to Property. The default is 150e3 Hertz.

PassbandRipple

Passband ripple of cascade response in dB.

Set this property to a double precision, positive scalar. When you set the `MinimumOrderDesign` property to true, the object designs the filters so that the cascade response meets the passband ripple that you specify in the `PassbandRipple` property. This property applies when you set the `FilterSpecification` property to `Design parameters` and the `MinimumOrderDesign` property to true. The default is `0.1` dB.

StopbandAttenuation

Stopband attenuation of cascade response in dB.

Set this property to a double precision, positive scalar. When you set the `MinimumOrderDesign` property to true, the object designs the filters so that the cascade response meets the stopband attenuation that you specify in the `StopbandAttenuation` property. This property applies when you set the `FilterSpecification` property to `Design parameters` and the `MinimumOrderDesign` property to true. The default is 60 dB.

Oscillator

Type of oscillator.

Specify the oscillator as one of `Sine wave` | `NCO`. The default is `Sine wave`. When you set this property to `Sine wave`, the object frequency up converts the output of the interpolation filter cascade using a complex exponential signal obtained from samples of a sinusoidal trigonometric function. When you set this property to `NCO` the object performs frequency up conversion with a complex exponential obtained using a numerically controlled oscillator (NCO).

CenterFrequency

Center frequency of output signal in Hertz .

Specify this property as a double precision, positive scalar. The value of this property must be less than or equal to half the product of the `SampleRate` property times the total interpolation

factor. The object up converts the input signal so that the output spectrum centers at the frequency you specify in the CenterFrequency property. The default is 14e6 Hertz.

NumAccumulatorBits

Number of NCO accumulator bits

Specify this property as an integer scalar in the range [1 128]. This property applies when you set the Oscillator property to NCO. The default is 16.

See also dsp.NCO.

NumQuantizedAccumulatorBits

Number of NCO quantized accumulator bits.

Specify this property as an integer scalar in the range[1 128]. The value you specify for this property must be less than the value you specify in the NumAccumulatorBits property. This property applies when you set the Oscillator property to NCO. The default is 12.

See also dsp.NCO.

Dither

Dither control for NCO .

When you set this property to true, the object uses the number of dither bits specified in the NumDitherBits property when applying dither to the NCO signal. This property applies when you set the Oscillator property to NCO. The default is true.

See also dsp.NCO.

NumDitherBits

Number of NCO dither bits.

Specify this property as an integer scalar smaller than the number of accumulator bits that you specify in the NumAccumulatorBits

property. This property applies when you set the Oscillator property to NCO and the Dither property to true. The default is 4.

See also dsp.NCO.

Fixed-Point Properties

FirstFilterCoefficientsDataType

Data type of first filter coefficients

Specify first filter coefficients data type as Same as input|Custom. The default is Same as input. This property applies when you set the FilterSpecification property to Coefficients.

CustomFirstFilterCoefficientsDataType

Fixed-point data type of first filter coefficients

Specify the first filter coefficients fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the FirstFilterCoefficientsDataType property to Custom. The default is numerictype([],16,15).

SecondFilterCoefficientsDataType

Data type of second filter coefficients

Specify the second filter coefficients data type as Same as input |Custom. The default is Same as input. This property applies when you set the FilterSpecification property to Coefficients.

CustomSecondFilterCoefficientsDataType

Fixed-point data type of second filter coefficients

Specify the second filter coefficients fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the SecondFilterCoefficientsDataType property to Custom. The default is numerictype([],16,15).

FiltersOutputDataType

Data type of output of each filter stage

Specify the data type at the output of the first (if it has not been bypassed), second, and third filter stages as one of `Same as input` | `Custom`. The default is `Same as input`. The object casts the data at the output of each filter stage according to the value you set in this property. For the CIC stage, the casting is done after the signal has been scaled by the normalization factor.

CustomFiltersOutputDataType

Fixed-point data type of output of each filter stage

Specify the filters output fixed-point type as a scaled numerictype object with a Signedness of `Auto`. This property applies when you set the `FiltersOutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

OutputDataType

Data type of output

Specify the data type of output as `Same as input` | `Custom`. The default is `Same as input`.

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a scaled numerictype object with a Signedness of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

Methods

clone	Create digital up converter object with same property values
fvtool	Visualize response of filter cascade

getFilterOrders	Get orders of interpolation filters
getFilters	Get handles to interpolation filter objects
getInterpolationFactors	Get interpolation factors of each filter stage
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
groupDelay	Group delay of filter cascade
isLocked	Locked status for input attributes and non-tunable properties
release	Allow property value and input characteristics changes
step	Process inputs using the object algorithm
visualizeFilterStages	Display response of filter stages

Examples   Create a digital up converter object that up samples a 1 KHz sinusoidal signal by a factor of 20 and up converts it to 50 KHz.

```
% Create a sine wave generator to obtain the 1 KHz sinusoidal signal
% with a sample rate of 6 KHz.
Fs = 6e3; % Sample rate
hSig = dsp.SineWave('Frequency',1000,'SampleRate', Fs,'SamplesPerFram
x = step(hSig); % generate signal

% Create a DUC object. Use minimum order filter designs and set the
% passband ripple to 0.2 dB and the stopband attenuation to 55 dB. Se
% the double sided signal bandwidth to 2 KHz.
hDUC = dsp.DigitalUpConverter(...
 'InterpolationFactor', 20,...
 'SampleRate', Fs,...
 'Bandwidth', 2e3,...
```

```
    'StopbandAttenuation', 55,...
    'PassbandRipple',0.2,...
    'CenterFrequency',50e3);

% Create a spectrum estimator to visualize the signal spectrum bet
% and after up converting.
hSEst = spectrum.welch; hSEst.SegmentLength = length(x)/10;
hOpts=psdopts(hSEst);
set(hOpts, 'Fs', Fs, 'SpectrumType','twosided','CenterDC',true);
figure; psd(hSEst,x,hOpts)
title('Spectrum of baseband signal x')

% Up convert the signal and visualize the spectrum
xUp = step(hDUC,x); % up convert
hOpts.Fs = Fs*20; hSEst.SegmentLength = length(xUp)/10;
figure; psd(hSEst,xUp,hOpts)
title('Spectrum of up converted signal xUp')

% Visualize the response of the interpolation filters
visualizeFilterStages(hDUC)
```

For additional examples using this System object, see the following demos:

- Digital Up and Down Conversion for Family Radio Service
- Design and Analysis of a Digital Down Converter

See Also   dsp.DigitalDownConverter

dsp.DigitalUpConverter.clone

Purpose   Create digital up converter object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a DigitalUpConverter System object, C, with the same property values as H.

The clone method creates a new unlocked object with uninitialized states.

Purpose   Visualize response of filter cascade

Syntax   fvtool(H)
               fvtool(H,..,'Arithmetic',ARITH,...)
               fvtool(H,..., PROP1, VALUE1,PROP2,VALUE2)

Description   fvtool(H) plots the magnitude response of the cascade of filters. By
                 default, the object plots the cascade response up to the second CIC null
                 frequency (or to the first when only one CIC null exists). When you set
                 the FilterSpecification property to Design parameters the method
                 plots a mask based on the filter specifications.

                 fvtool(H,..,'Arithmetic',ARITH,...) specifies the arithmetic
                 of the filter cascade. You set input ARITH to double, single, or
                 fixed-point. When object H is in an unlocked state you must specify
                 the arithmetic. When object H is in a locked state the arithmetic input
                 is ignored.

                 fvtool(H,..., PROP1, VALUE1,PROP2,VALUE2) launches FVTool and
                 sets the specified FVTool properties to the specified values.

dsp.DigitalUpConverter.getFilters

Purpose   Get handles to interpolation filter objects

Syntax   `S = getFilters(H)`
`getFilters(H,'Arithmetic',ARITH)`

Description   `S = getFilters(H)` returns a structure, `S`, with copies of the filter System objects and the CIC normalization factor that form the interpolation filter cascade. The `FirstFilterStage` structure field is empty if the first filter stage has been bypassed. The CIC normalization factor equals the inverse of the CIC filter gain. In some cases, this gain includes a correction factor to ensure that the cascade response meets the ripple specifications.

`getFilters(H,'Arithmetic',ARITH)` specifies the arithmetic of the filter stages. You set `ARITH` to `double`, `single`, or `fixed-point`. When object `H` is in an unlocked state, you must specify the arithmetic input. When object `H` is in a locked state, the arithmetic input is ignored.

When `H` is in an unlocked state, and you specify the arithmetic as `fixed-point`, the `getFilters` method returns filter System objects. The custom coefficient data type properties of these System objects are set to the values that the `dsp.DigitalUpConverter` System object uses to process data when you call the `step` method. All other fixed-point properties are set to their default values.

When `H` is in a locked state, and the input to the step method is of a fixed-point data type, the `getFilters` method returns filter System objects. All fixed-point properties of these System objects are set to the exact values that the `dsp.DigitalUpConverter` System object uses to process the data.

Purpose   Get orders of interpolation filters

Syntax   S = getFilterOrders(H)

Description   S = getFilterOrders(H) returns a structure, S, that contains the
                    orders of the interpolation filter stages. The FirstFilterOrder
                    structure field will be empty if the first filter stage has been bypassed.

dsp.DigitalUpConverter.getInterpolationFactors

Purpose   Get interpolation factors of each filter stage

Syntax   M = getInterpolationFactors(H

Description M = getInterpolationFactors(H) returns a vector, M, with the
                interpolation factors of each filter stage. If the first filter stage is
                bypassed, then M is a 1-by-2 vector containing the interpolation
                factors of the second and third stages in the first and second elements
                respectively. If the first filter stage is not bypassed then M is a 1-by-3
                vector containing the interpolation factors of the first, second and third
                filter stages.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                     the step method

                     The getNumInputs method returns a positive integer that is the number
                     of expected inputs (not counting the object itself) to the step method.
                     This value will change if you alter any properties that turn inputs on or
                     off. You must call the step method with the number of input arguments
                     equal to the result of getNumInputs(*H*).

dsp.DigitalUpConverter.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
method

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

Purpose   Group delay of filter cascade

Syntax   D = groupDelay(H,N)
                   [D,F] = groupDelay(H,N)

Description   D = groupDelay(H,N) returns a vector of group delays, D, evaluated
                   at N frequency points equally spaced around the upper half of the unit
                   circle. If you don't specify N, it defaults to 8192.

                   [D,F] = groupDelay(H,N) returns a vector, F, of frequencies at which
                   the group delay has been computed.

dsp.DigitalUpConverter.isLocked

Purpose   Locked status for input attributes and non-tunable properties

Syntax   Y = isLocked(H)

Description Y = isLocked(H) returns the locked state, Y, of the
                DigitalUpConverter object.

                The isLocked method returns a logical value that indicates whether
                input attributes and nontunable properties for the object are locked. The
                object performs an internal initialization the first time the step method
                is executed. This initialization locks nontunable properties and input
                specifications, such as dimensions, complexity, and data type of the
                input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections. This method lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.DigitalUpConverter.step

Purpose   Process inputs using the object algorithm

Syntax   Y = step(H,X)

Description   Y = step(H,X) takes a real or complex input column vector X and
                outputs an up sampled and frequency up converted signal Y. X can be
                of data type double, single, signed integer, or signed fixed point (fi
                objects). The length of Y is equal to the length of X multiplied by the
                InterpolationFactor. When the data type of X is double or single
                precision, the data type of Y is the same as that of X. When the data
                type of X is of a fixed point type, the data type of Y is defined by the
                OutputDataType property.

Purpose   Display response of filter stages

Syntax   visualizeFilterStages(H)
visualizeFilterStages(H,'Arithmetic',ARITH)
hfvt = visualizeFilterStages(H)

Description visualizeFilterStages(H) plots the magnitude response of the filter
stages and of the cascade response. When the FilterSpecification
property is set to Design parameters the method plots a mask based
on the filter specifications. By default, the object plots the response of
the filters up to the second CIC null frequency (or to the first when
only one CIC null exists).

visualizeFilterStages(H,'Arithmetic',ARITH) specifies the
arithmetic of the filter stages. You set input ARITH to 'double', 'single',
or 'fixed-point'. When object H is in an unlocked state you must specify
the arithmetic. When object H is in a locked state the arithmetic input
is ignored.

hfvt = visualizeFilterStages(H) returns the handle to the FVTool
object.

dsp.DyadicAnalysisFilterBank

Purpose   Dyadic analysis filter bank

Description   The `DyadicAnalysisFilterBank` object uses a series of highpass and lowpass FIR filters to provide approximate octave band frequency decompositions of the input. Each filter output is downsampled by a factor of two. With the appropriate analysis filters and tree structure, the dyadic analysis filter bank is a discrete wavelet transform (DWT) or discrete wavelet packet transform (DWPT).

To obtain approximate octave band frequency decompositions of the input:

1 Define and set up your dyadic analysis filter bank. See "Construction" on page 5-414.

2 Call `step` to get the octave half band frequency decompositions of the input according to the properties of `dsp.DyadicAnalysisFilterBank`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.DyadicAnalysisFilterBank` constructs a dyadic analysis filter bank object, `H`, that computes the level-two discrete wavelet transform (DWT) of a column vector input. For a 2-D matrix input, the object transforms the columns using the Daubechies third-order extremal phase wavelet. The length of the input along the first dimension must be a multiple of 4.

`H = dsp.DyadicAnalysisFilterBank('`*PropertyName*`',`*PropertyValue*`, ...)` returns a dyadic analysis filter bank object, `H`, with each property set to the specified value.

Properties   **Filter**

Type of filter used in subband decomposition

Specify the type of filter used to determine the high and lowpass FIR filters in the dyadic analysis filter bank as `Custom`, `Haar`, `Daubechies`, `Symlets`, `Coiflets`, `Biorthogonal`, `Reverse`

Biorthogonal, or Discrete Meyer. All property values except Custom require Wavelet Toolbox software. If the value of this property is Custom, the filter coefficients are specified by the values of the CustomLowpassFilter and CustomHighpassFilter properties. Otherwise, the dyadic analysis filter bank object uses the Wavelet Toolbox function wfilters to construct the filters. The following table lists supported wavelet filters and example syntax to construct the filters:

Filter	Example Setting	Syntax for Analysis Filters
Haar	N/A	[Lo_D,Hi_D]=wfilters('haar');
Daubechies extremal phase	WaveletOrder=3;	[Lo_D,Hi_D]=wfilters('db3');
Symlets (Daubechies least-asymmetric)	WaveletOrder=4;	[Lo_D,Hi_D]=wfilters('sym4');
Coiflets	WaveletOrder=1;	[Lo_D,Hi_D]=wfilters('coif1');
Biorthogonal	FilterOrder='[3/1]';	[Lo_D,Hi_D,Lo_R,Hi_R]=... wfilters('bior3.1');
Reverse biorthogonal	FilterOrder='[3/1]';	[Lo_D,Hi_D,Lo_R,Hi_R]=... wfilters('rbior3.1');
Discrete Meyer	N/A	[Lo_D,Hi_D]=wfilters('dmey');

CustomLowpassFilter

Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in powers of z^{-1}. Use a half-band filter that passes the frequency band stopped by the filter specified in the CustomHighpassFilter property. This property applies when you set the Filter property to Custom. The default specifies a Daubechies third-order extremal phase scaling (lowpass) filter.

CustomHighpassFilter

Highpass FIR filter coefficients

Specify a vector of highpass FIR filter coefficients, in powers of z^{-1}. Use a half-band filter that passes the frequency band stopped by the filter specified in the CustomLowpassFilter property. This property applies when you set the Filter property to Custom. The default specifies a Daubechies 3rd-order extremal phase wavelet (highpass) filter.

WaveletOrder

Order for orthogonal wavelets

Specify the order of the wavelet selected in the Filter property. This property applies when you set the Filter property to an orthogonal wavelet: Daubechies (Daubechies extremal phase), Symlets (Daubechies least-asymmetric), or Coiflets. The default is 2.

FilterOrder

Analysis and synthesis filter orders for biorthogonal filters

Specify the order of the analysis and synthesis filter orders for biorthogonal filter banks as 1 / 1, 1 / 3, 1 / 5, 2 / 2, 2 / 4, 2 / 6, 2 / 8, 3 / 1, 3 / 3, 3 / 5, 3 / 7, 3 / 9, 4 / 4, or 5 / 5, 6 / 8. Unlike orthogonal wavelets, biorthogonal wavelets require different filters for the analysis (decomposition) and synthesis (reconstruction) of an input. The first number indicates the order of the synthesis (reconstruction) filter. The second number indicates the order of the analysis (decomposition) filter. This property applies when you set the Filter property to Biorthogonal or Reverse Biorthogonal. The default is 1 / 1.

NumLevels

Number of filter bank levels used in analysis (decomposition)

Specify the number of filter bank analysis levels a positive integer. A level-N asymmetric structure produces $N+1$ output subbands. A level-N symmetric structure produces 2^N output subbands. The

default is 2. The size of the input along the first dimension must be a multiple of 2^N, where N is the number of levels.

TreeStructure

Structure of filter bank

Specify the structure of the filter bank as Asymmetric or Symmetric. The asymmetric structure decomposes only the lowpass filter output from each level. The symmetric structure decomposes the highpass and lowpass filter outputs from each level. If the analysis filters are scaling (lowpass) and wavelet (highpass) filters, the asymmetric structure is the discrete wavelet transform, while the symmetric structure is the discrete wavelet packet transform.

When this property is Symmetric, the output has 2^N subbands each of size $M/2^N$. In this case, M is the length of the input along the first dimension and N is the value of the NumLevels property. When this property is Asymmetric, the output has $N+1$ subbands. The following equation gives the length of the output in the kth subband in the asymmetric case:

$$M_k = \begin{cases} \dfrac{M}{2^k} & 1 \le k \le N \\ \dfrac{M}{2^N} & k = N+1 \end{cases}$$

The default is Asymmetric.

Methods

clone	Create dyadic analysis filter bank object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method

dsp.DyadicAnalysisFilterBank

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset filter states
step	Decompose input with dyadic filter bank

Examples

Denoise square wave input using dyadic analysis and synthesis filter banks:

```
t = 0:.0001:.0511;
x= square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1);
hdydanl = dsp.DyadicAnalysisFilterBank;

% The filter coefficients correspond to a 'haar' wavelet.
hdydanl.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];
hdydanl.CustomHighpassFilter = [-1/sqrt(2) 1/sqrt(2)];
hdydsyn = dsp.DyadicSynthesisFilterBank;
hdydsyn.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];
hdydsyn.CustomHighpassFilter = [1/sqrt(2) -1/sqrt(2)];

C = step(hdydanl, xn);

% Subband outputs
C1 = C(1:256); C2 = C(257:384); C3 = C(385:512);

% Set higher frequency coefficients to zero
% to remove the noise.
x_den = step(hdydsyn, [zeros(length(C1),1);...
zeros(length(C2),1);C3]);

subplot(2,1,1), plot(xn); title('Original noisy Signal');
subplot(2,1,2), plot(x_den); title('Denoised Signal');
```

Algorithms   This object implements the algorithm, inputs, and outputs described
on the Dyadic Analysis Filter Bank block reference page. The object
properties correspond to the block parameters, except:

The dyadic analysis filter bank object always concatenates the
subbands into a single column vector for a column vector input,
or into the columns of a matrix for a matrix input. This behavior
corresponds to the block's behavior when you set the **Output**
parameter to Single port.

See Also   dsp.DyadicSynthesisFilterBank | dsp.SubbandAnalysisFilter

Purpose   Create dyadic analysis filter bank object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a dyadic analysis filter bank object, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose Number of expected inputs to step method

Syntax N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

dsp.DyadicAnalysisFilterBank.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the dyadic analysis filter bank
                   object.

                   The isLocked method returns a logical value that indicates whether
                   input attributes and nontunable properties for the object are locked. The
                   object performs an internal initialization the first time the step method
                   is executed. This initialization locks nontunable properties and input
                   specifications, such as dimensions, complexity, and data type of the
                   input data. After locking, the isLocked method returns a true value.

dsp.DyadicAnalysisFilterBank.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset filter states

Syntax   reset(H)

Description   reset(H) resets the filter states of the dyadic analysis filter bank
                   object, H, to their initial values of zero. After the step method applies
                   the dyadic analysis filter bank to nonzero input data, the filter states
                   may change. Invoking the step method again without first invoking the
                   reset method may produce different outputs for an identical input.

                   Example of resetting filter states:

```
X = [1 1 7 9 2 8 8 6]';
H = dsp.DyadicAnalysisFilterBank('NumLevels',1);
% Filter states are zero
y = step(H,X);
% Invoke step method again without resetting states
y1 = step(H,X);
isequal(y,y1) % Returns 0
reset(H); % Reset filter states to zero
y2 = step(H,X);
isequal(y,y2) % Returns 1
```

dsp.DyadicAnalysisFilterBank.step

Purpose	Decompose input with dyadic filter bank
Syntax	Y = step(H,X)
Description	Y = step(H,X) computes the subband decomposition of the input X and outputs the dyadic subband decomposition in Y as a single concatenated column vector or matrix of coefficients. Each column of X is treated as an independent input, and the number of rows of X must be a multiple of 2^N, where N is the value of the NumLevels property. The elements of Y are ordered with the highest frequency subband first followed by subbands in decreasing frequency

Examples   Subband ordering for level-two asymmetric tree structure:

```
t = 0:.001:1.023;
% Sampling frequency 1 kHz input length 1024
x= square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1);
% Default asymmetric structure with
% Daubechies order 3 extremal phase wavelet
H = dsp.DyadicAnalysisFilterBank;
Y = step(H,xn);
% Level 2 yields 3 subbands (two detail-one approximation)
% Nyquist frequency is 500 Hz
D1 =Y(1:512); % subband approx. [250, 500] Hz
D2 = Y(513:768); % subband approx. [125, 250] Hz
Approx = Y(769:1024); % subband approx. [0,125] Hz
```

Subband ordering for symmetric tree structure:

```
t = 0:.001:1.023;
% Sampling frequency 1 kHz input length 1024
x= square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1); % symmetric structure with
% Daubechies order 3 extremal phase wavelet
```

```
H = dsp.DyadicAnalysisFilterBank('TreeStructure',...
'Symmetric');
Y = step(H,xn);
D1 = Y(1:256); % subband approx. [375,500] Hz
D2 = Y(257:512); % subband approx. [250,375] Hz
D3 = Y(513:768); % subband approx. [125,250] Hz
Approx = Y(769:1024); % subband approx. [0, 125] Hz
```

dsp.DyadicSynthesisFilterBank

Purpose   Reconstruct signals from subbands

Description   The `DyadicSynthesisFilterBank` object reconstructs signals from subbands with smaller bandwiths and lower sample rates. The filter bank uses a series of highpass and lowpass FIR filters to repeatedly reconstruct the signal.

To reconstruct signals from subbands with smaller bandwidths and lower sample rates:

1 Define and set up your synthesis filter bank. See "Construction" on page 5-428.

2 Call `step` to reconstruct the signal according to the properties of `dsp.DyadicSynthesisFilterBank`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.DyadicSynthesisFilterBank` returns a synthesis filter bank, `H`, that reconstructs a signal from its subbands with smaller bandwidths and smaller sample rates.

`H = dsp.DyadicSynthesisFilterBank('`*Propertyname*`',`*PropertyValue*`,...)` returns a synthesis filter bank, `H`, with each specified property set to the specified value.

Properties   **Filter**

Type of filter used in filter bank

Specify the type of filter used to determine the highpass and lowpass FIR filters in the filter bank as one of `Custom`, `Haar`, `Daubechies`, `Symlets`, `Coiflets`, `Biorthogonal`, `Reverse Biorthogonal`, or `Discrete Meyer`. If you set this property to `Custom`, the `CustomLowpassFilter` and `CustomHighpassFilter` properties specify the filter coefficients. Otherwise, the object uses the Wavelet Toolbox `wfilters` function to construct the filters. Depending on the filter, the `WaveletOrder` or `FilterOrder`

property might apply. For a list of the supported wavelets, see the following table.

Filter	Sample Setting for Related Filter Specification Properties	Corresponding Wavelet Toolbox Function Syntax
Haar	None	wfilters('haar')
Daubechies	H.WaveletOrder = 4	wfilters('db4')
Symlets	H.WaveletOrder = 3	wfilters('sym3')
Coiflets	H.WaveletOrder = 1	wfilters('coif1')
Biorthogonal	H.FilterOrder = '[3/1]'	wfilters('bior3.1')
Reverse Biorthogonal	H.FilterOrder = '[3/1]'	wfilters('rbior3.1')
Discrete Meyer	None	wfilters('dmey')

In order to automatically design wavelet-based filters, install the Wavelet Toolbox product. Otherwise, use the `CustomLowpassFilter` and `CustomHighpassFilter` properties to specify lowpass and highpass FIR filters.

CustomLowpassFilter

Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in descending powers of *z*. Use a half-band filter that passes the frequency band stopped by the filter specified in the `CustomHighpassFilter` property. This property applies only when you set the `Filter` property to `Custom`. To perfectly reconstruct a signal decomposed by the `DyadicAnalysisFilterBank`, design the filters in the

synthesis filter bank to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is imperfect. The default values of this property specify a perfect reconstruction filter for the default settings of the analysis filter bank (based on a third-order Daubechies wavelet).

CustomHighpassFilter

Highpass FIR filter coefficients

Specify a vector of highpass FIR filter coefficients, in descending powers of *z*. Use a half-band filter that passes the frequency band stopped by the filter specified in the `CustomLowpassFilter` property. This property applies only when you set the `Filter` property to `Custom`. To perfectly reconstruct a signal decomposed by the `DyadicAnalysisFilterBank`, design the filters in the synthesis filter bank to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction is imperfect. The default values of this property specify a perfect reconstruction filter for the default settings of the analysis filter bank (based on a third-order Daubechies wavelet).

WaveletOrder

Wavelet order

Specify the order of the wavelet selected in the `Filter` property. This property applies only when you set the `Filter` property to `Daubechies`, `Symlets` or `Coiflets`. The default is 2.

FilterOrder

Wavelet order for synthesis filter stage

Specify the order of the wavelet for the synthesis filter stage as:

- First order: `'[1/1]'`, `'[1/3]'`, or `'[1/5]'`.
- Second order: `'[2/2]'`, `'[2/4]'`, `'[2/6]'`, or `'[2/8]'`.
- Third order: `'[3/1]'`, `'[3/3]'`, `'[3/5]'`, `'[3/7]'`, or `'[3/9]'`.
- Fourth order: `'[4/4]'`.

- Fifth order: `'[5/5]'`.

- Sixth order: `'[6/8]'`.

This property applies only when you set the `Filter` property to `Biorthogonal` or `Reverse Biorthogonal`. The default is `'[1/1]'`.

NumLevels

Number of filter bank levels

Specify the number of filter bank levels as a scalar integer. An N-level asymmetric structure has $N + 1$ input subbands, and an N-level symmetric structure has 2^N input subbands. The default is 2.

rg

TreeStructure

Structure of filter bank

Specify the structure of the filter bank as `Asymmetric` or `Symmetric`. In the asymmetric structure, the low-frequency subband input to each level is the output of the previous level, while the high-frequency subband input to each level is an input to the filter bank. In the symmetric structure, both the low- and high-frequency subband inputs to each level are outputs from the previous level. The default is `Asymmetric`.

Methods		
	clone	Create synthesis filter bank object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of the step method
	isLocked	Locked status for input attributes and nontunable properties

dsp.DyadicSynthesisFilterBank

release	Allow property value and input characteristics changes
reset	Reset internal states of dyadic synthesis filter bank object
step	Reconstruct signal from high- and low-frequency subbands

Examples   Remove noise from a signal using filter banks:

```
t = 0:.0001:.0511;
x= square(2*pi*30*t);
xn = x' + 0.08*randn(length(x),1);
hdydanl = dsp.DyadicAnalysisFilterBank;

% The filter coefficients correspond to a 'haar' wavelet.
hdydanl.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];
hdydanl.CustomHighpassFilter = [-1/sqrt(2) 1/sqrt(2)];
hdydsyn = dsp.DyadicSynthesisFilterBank;
hdydsyn.CustomLowpassFilter = [1/sqrt(2) 1/sqrt(2)];
hdydsyn.CustomHighpassFilter = [1/sqrt(2) -1/sqrt(2)];

C = step(hdydanl, xn);

% Subband outputs
C1 = C(1:256); C2 = C(257:384); C3 = C(385:512);

% Set high-frequency coefficients to zero to remove noise.
x_den = step(hdydsyn, ...
  [zeros(length(C1),1); ...
   zeros(length(C2),1); ...
   C3]);

subplot(2,1,1), plot(xn); title('Original Noisy Signal');
subplot(2,1,2), plot(x_den); title('Denoised Signal');
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Dyadic Synthesis Filter Bank block reference page. The object properties correspond to the block parameters, except:

The object only receives data as a vector or matrix of concatenated subbands, as specified using the `step` method.

See Also   dsp.DyadicAnalysisFilterBank | dsp.SubbandSynthesisFilter

dsp.DyadicSynthesisFilterBank.clone

Purpose   Create synthesis filter bank object with same property values

Syntax   `C = clone(H)`

Description   `C = clone(H)` creates a `DyadicSynthesisFilterBank` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

dsp.DyadicSynthesisFilterBank.getNumOutputs

Purpose   Number of outputs of the step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the
DyadicSynthesisFilterBank object.

The isLocked method returns a logical value that indicates whether
input attributes and nontunable properties for the object are locked. The
object performs an internal initialization the first time the step method
is executed. This initialization locks nontunable properties and input
specifications, such as dimensions, complexity, and data type of the
input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset internal states of dyadic synthesis filter bank object

Syntax   reset(H)

Description   reset(H) sets the internal states of the dyadic synthesis filter bank object H to their initial values.

Purpose   Reconstruct signal from high- and low-frequency subbands

Syntax   X = step(H,S)

Description   X = step(H,S) reconstructs the concatenated subband input S to output X. Each column of input S contains the subbands for an independent signal. Upper rows contain the high-frequency subbands, and lower rows contain the low-frequency subbands. The number of rows of S must be a multiple of 2^N, where *N* is the value of the NumLevels property.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Discrete Fourier transform

Description   The FFT object computes the discrete Fourier transform (DFT) of an input. The object uses one or more of the following fast Fourier transform (FFT) algorithms depending on the complexity of the input and whether the output is in linear or bit-reversed order:

- Double-signal algorithm

- Half-length algorithm

- Radix-2 decimation-in-time (DIT) algorithm

- Radix-2 decimation-in-frequency (DIF) algorithm

- An algorithm chosen by FFTW [1], [2]

To compute the DFT of an input:

1 Define and set up your FFT object. See "Construction" on page 5-441.

2 Call step to compute the DFT of the input according to the properties of dsp.FFT. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.FFT returns a FFT object, H, that computes the DFT of an *N*-D array. For column vectors or multidimensional arrays, the FFT object computes the DFT along the first dimension. If the input is a row vector, the FFT object computes a row of single-sample DFTs and issues a warning.

H = dsp.FFT('*PropertyName*',*PropertyValue*, ...) returns a FFT object, H, with each property set to the specified value.

Properties   **FFTImplementation**

FFT implementation

Specify the implementation used for the FFT as one of `Auto` | `Radix-2` | `FFTW`. When you set this property to `Radix-2`, the FFT length must be a power of two.

BitReversedOutput

Order of output elements relative to input elements

Designate order of output channel elements relative to order of input elements. Set this property to `true` to output the frequency indices in bit-reversed order. The default is `false`, which corresponds to a linear ordering of frequency indices.

Normalize

Divide butterfly outputs by two

Set this property to `true` if the output of the FFT should be divided by the FFT length. This option is useful when you want the output of the FFT to stay in the same amplitude range as its input. This is particularly useful when working with fixed-point data types.

The default value of this property is `false` with no scaling.

FFTLengthSource

Source of FFT length

Specify how to determine the FFT length as `Auto` or `Property`. When you set this property to `Auto`, the FFT length equals the number of rows of the input signal. The default is `Auto`.

FFTLength

FFT length

Specify the FFT length. This property applies when you set the `FFTLengthSource` property to `Property`. The default is `64`.

This property must be a power of two when the input is a fixed-point data type, or when you set the `BitReversedOutput` property to `true`, or when you set the `FFTImplementation` property to `Radix-2`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

SineTableDataType

Sine table word and fraction lengths

Specify the sine table data type as `Same word length as input` or `Custom`. The default is `Same word length as input`.

CustomSineTableDataType

Sine table word and fraction lengths

Specify the sine table fixed-point type as an unscaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineTableDataType` property to `Custom`. The default is `numerictype([],16)`.

ProductDataType

Product word and fraction lengths

Specify the product data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when

you set the ProductDataType property to Custom. The default is numerictype([],32,30).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator data type as Full precision, Same as input, Same as product, or Custom. The default is Full precision.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the AccumulatorDataType property to Custom. The default is numerictype([],32,30).

OutputDataType

Output word and fraction lengths

Specify the output data type as one of Full precision, Same as input, Custom. The default is Full precision.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the OutputDataType property to Custom. The default is numerictype([],16,15).

Methods

clone	Create FFT object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Discrete Fourier transform of input

Examples   Find frequency components of a signal in additive noise:

```
Fs = 800; L = 1000;
t = (0:L-1)'/Fs;
x = sin(2*pi*250*t) + 0.75*cos(2*pi*340*t);
y = x + .5*randn(size(x)); % noisy signal

hfft = dsp.FFT('FFTLengthSource', 'Property', ...
 'FFTLength', 1024);

Y = step(hfft, y);

% Plot the single-sided amplitude spectrum
plot(Fs/2*linspace(0,1,512), 2*abs(Y(1:512)/1024));
title('Single-sided amplitude spectrum of noisy signal y(t)');
xlabel('Frequency (Hz)'); ylabel('|Y(f)|');
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the FFT block reference page. The object properties correspond to the block parameters.

References   [1] FFTW (http://www.fftw.org)

[2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

See Also   dsp.IFFT | dsp.DCT | dsp.IDCT

Purpose   Create FFT object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an FFT object with the same property values as H.
                     The clone method creates a new unlocked object.

dsp.FFT.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

dsp.FFT.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax isLocked(H)

Description isLocked(H) returns the locked state of the FFT object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.FFT.step

Purpose Discrete Fourier transform of input

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` computes the FFT, `Y` , of the input `X` along the first dimension of `X`. When the `FFTLengthSource` property is `Auto`, the length of `X` along the first dimension must be a positive integer power of two. This length is also the FFT length . When the `FFTLengthSource` property is `Property`, the `FFTLength` property must be a positive integer power of two.

Purpose   FIR polyphase decimator

Description   The FIRDecimator object resamples vector or matrix inputs along
                  the first dimension. The object reseamples at a rate *M* times
                  slower than the input sampling rate, where *M* is the integer-valued
                  downsampling factor. The decimation combines an FIR anti-aliasing
                  filter with downsampling. The FIR decimator object uses a polyphase
                  implementation of the FIR filter.

                  To resample vector or matrix inputs along the first dimension:

                  1 Define and set up your FIR decimator. See "Construction" on page
                  5-453.

                  2 Call step to resample the vector or matrix inputs according to the
                  properties of dsp.FIRDecimator. The behavior of step is specific to
                  each object in the toolbox.

Construction H = dsp.FIRDecimator returns an FIR decimator, H, which applies an
                  FIR filter with a cutoff frequency of 0.4*pi radians/sample to the input
                  and downsamples the filter output by factor of 2.

                  H = dsp.FIRDecimator ('*PropertyName*',*PropertyValue*, ...)
                  returns an FIR decimator, H, with each property set to the specified
                  value.

                  H = dsp.FIRDecimator(DECIM, NUM,
                  '*PropertyName*',*PropertyValue*, ...) returns an FIR decimator, H,
                  with the integer-valued DecimationFactor property set to DECIM, the
                  Numerator property set to NUM, and other specified properties set to
                  the specified values.

Properties   **DecimationFactor**

                  Decimation factor

                  Specify the downsampling factor as a positive integer. The FIR
                  decimator reduces the sampling rate of the input by this factor.

The size of the input along the first dimension must be a multiple of the decimation factor. The default is 2.

Numerator

FIR filter coefficients

Specify the numerator coefficients of the FIR filter in powers of z^{-1}. The following equation defines the system function for a filter of length L:

$$H(z) = \sum_{l=0}^{L-1} b_l z^{-l}$$

To prevent aliasing as a result of downsampling, the filter transfer function should have a normalized cutoff frequency no greater than 1/DecimationFactor. You can specify the filter coefficients as a vector in the supported data types. The FIR decimator does not support dfilt or mfilt objects as sources of the filter coefficients. The default is fir1(35,0.4).

Structure

Filter structure

Specify the implementation of the FIR filter as either Direct form or Direct form transposed. The default is Direct form.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are

added, as needed, to ensure that no roundoff or overflow occurs. If you set FullPrecisionOverride to false, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of Wrap or Saturate. This property applies only if the object is not in full precision mode.

CoefficientsDataType

Coefficients word and fraction lengths

Specify the coefficients fixed-point data type as Same word length as input or Custom. The default is Same word length as input.

CustomCoefficientsDataType

Coefficients word and fraction lengths

Specify the coefficients fixed-point type as a numerictype object with a Signedness of Auto. This property applies when you set the CoefficientsDataType property to Custom. The default is numerictype([],16,15).

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | `Full precision` | `Same as product` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | `Same as accumulator` | `Same as product` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when

you set the `OutputDataType` property to `Custom`. The default is `numerictype([]1,16,15)`.

	Methods
Methods	

clone	Create FIR decimator object with same property values
freqz	Frequency response
fvtool	Open filter visualization tool
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
impz	Impulse response
isLocked	Locked status for input attributes and nontunable properties
phasez	Unwrapped phase response
release	Allow property value and input characteristics changes
reset	Reset filter states of FIR decimator
step	Decimate input by integer factor

More "Analysis Methods for Filter System Objects" on page 4-2.

Definitions

Polyphase Subfilters

A polyphase implementation of an FIR decimator *splits* the lowpass FIR filter impulse response into *M* different subfilters, where *M* is the downsampling, or decimation factor. Let *h(n)* denote the FIR filter impulse response of length *L* and *u(n)* the input signal. Decimating the filter output by a factor of *M* is equivalent to the downsampled convolution:

dsp.FIRDecimator

$$y(n) = \sum_{l=0}^{L-1} h(l)u(nM - l)$$

The key to the efficiency of polyphase filtering is that specific input values are only multiplied by select values of the impulse response in the downsampled convolution. For example, letting *M=2*, the input values *u(0),u(2),u(4), ...* are only combined with the filter coefficients *h(0),h(2),h(4),...*, and the input values *u(1),u(3),u(5), ...* are only combined with the filter coefficients *h(1),h(3),h(5),....* By splitting the filter coefficients into two polyphase subfilters, no unnecessary computations are performed in the convolution. The outputs of the convolutions with the polyphase subfilters are interleaved and summed to yield the filter output. The following MATLAB code demonstrates how to construct the two polyphase subfilters for the default order 35 filter in the `Numerator` property and the default `DecimationFactor` property value of two:

```
M = 2;
Num = fir1(35,0.4);
FiltLength = length(Num);
Num = flipud(Num(:));

if (rem(FiltLength, M) ~= 0)
    nzeros = M - rem(FiltLength, M);
    Num = [zeros(nzeros,1); Num];  % Appending zeros
end

len = length(Num);
nrows = len / M;
PolyphaseFilt = flipud(reshape(Num, M, nrows).');
```

The columns of `PolyphaseFilt` are subfilters containing the two *phases* of the filter in `Num`. For a general downsampling factor of *M*, there are *M* phases and therefore *M* subfilters.

Examples   Decimate a sum of sine waves with angular frequencies of π/4 and 2π/3 radians/sample by a factor of two. To prevent aliasing, the

FIR decimator filters out the 2π/3 radians/sample component before downsampling:

```
x = cos(pi/4*[0:95]')+sin(2*pi/3*[0:95]');
H = dsp.FIRDecimator;
y = step(H,x);

% View group delay of default FIR filter
fvtool(fir1(35,0.4),1,'analysis','grpdelay');
% Group delay of the default linear-phase FIR filter
% is 17.5 samples. Downsampling by a factor of
% two expect an approx. 8.75 sample delay in the output
% y with the initial filter states of zero

subplot(211);
stem(x(1:length(x)/2),'b','markerfacecolor',[0 0 1]);
title('Input Signal');
subplot(212);
stem(y,'b','markerfacecolor',[0 0 1]);
title('Output--Lowpass filtered and downsampled by 2');
```

The figure shows that the delay in the decimated output is consistent with the group delay of the filter when the initial filter states are zero.

Reduce the sampling rate of an audio signal by 1/2 and play it:

```
hmfr = dsp.AudioFileReader('OutputDataType',...
  'single');
hap = dsp.AudioPlayer(22050/2);
hfirdec = dsp.FIRDecimator;
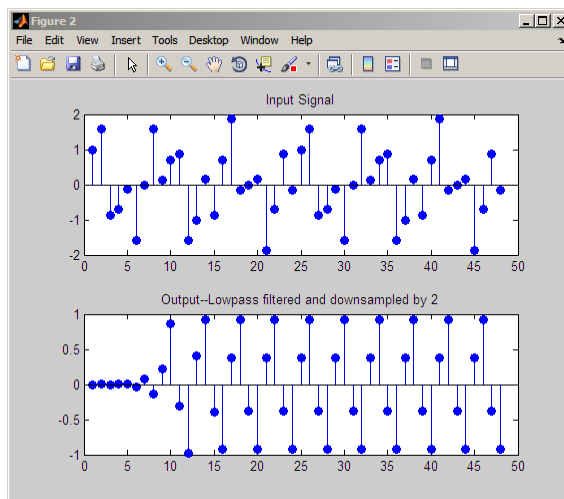
while ~isDone(hmfr)
    frame = step(hmfr);
    y = step(hfirdec, frame);
    step(hap, y);
end

release(hmfr);
pause(0.5);
release(hap);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the FIR Decimation block reference page. The object properties correspond to the block parameters, except:

- **Coefficient source** – The FIR decimator object does not support `mfilt` objects.

- **Framing** – The FIR decimator object only supports `Maintain input frame rate`

- **Output buffer initial conditions** – The FIR decimator object does not support this parameter.

- **Rate options** – The FIR decimator object does not support this parameter.

- **Input processing** The FIR decimator object does not support this parameter.

See Also   dsp.FIRInterpolator | dsp.FIRRateConverter

dsp.FIRDecimator.clone

Purpose   Create FIR decimator object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an FIR decimator object, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Frequency response

Syntax
```
[h,w] = freqz(H)
[h,w] = freqz(H,n)
[h,w] = freqz(H,Name,Value)
freqz(H)
```

Description `[h,w] = freqz(H)` returns the complex, 8192–element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample.

`[h,w] = freqz(H,n)` returns the complex, n-element frequency response vector `h`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[h,w] = freqz(H,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(H)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object `H`.

dsp.FIRDecimator.fvtool

Purpose   Open filter visualization tool

Syntax   fvtool(H)
                fvtool(H,'Arithmetic',ARITH,...)

Description fvtool(H) performs an analysis and computes the magnitude response
                of the filter System object H.

                fvtool(H,'Arithmetic',ARITH,...) analyzes the filter System object
                H, based on the arithmetic specified in the ARITH input. ARITH can
                be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a
                double precision filter when the arithmetic input is not specified and
                the filter System object is in an unlocked state. The 'Arithmetic' input
                is only relevant for the analysis of filter System objects.

Purpose   Number of expected inputs to step method

Syntax   `N = getNumInputs(H)`

Description   `N = getNumInputs(H)` returns the number of expected inputs, `N`, to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

dsp.FIRDecimator.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Impulse response

Syntax   `[h,t] = impz(H)`
                  `[h,t] = impz(H,Name,Value)`
                  `impz(H)`

Description   `[h,t] = impz(H)` returns the impulse response `h`, and the
                  corresponding time points `t` at which the impulse response of `H` is
                  computed.

                  `[h,t] = impz(H,Name,Value)` returns the impulse response `h`, and the
                  corresponding time points `t` , with additional options specified by one or
                  more Name, Value pair arguments.

                  `impz(H)` uses FVTool to plot the impulse response of the filter System
                  object `H`.

                  Note You can use `impz` for both real and complex filters. When you
                  omit the output arguments, `impz` plots only the real part of the impulse
                  response.

                  ---

dsp.FIRDecimator.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the FIR decimator.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Unwrapped phase response

Syntax
```
[phi,w] = phasez(H)
[phi,w] = phasez(H,n)
[phi,w] = phasez(H,Name,Value)
phasez(H)
```

Description   [phi,w] = phasez(H) returns the 8192–element phase response vector phi, and the corresponding frequencies w in radians/sample.

[phi,w] = phasez(H,n) returns the n-element phase response vector phi, and the corresponding frequencies w in radians/sample, using n samples.

[phi,w] = phasez(H,Name,Value) returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

phasez(H) uses FVTool to plot the phase response of the filter System object H.

dsp.FIRDecimator.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset filter states of FIR decimator

Syntax   `reset(H)`

Description `reset(H)` resets the filter states of the FIR decimator object,H, to the
initial values of zero. After the `step` method applies the FIR decimator
to nonzero input data, the filter states may change. Invoking the `step`
method again without first invoking the `reset` method may produce
different outputs for an identical input.

Example of resetting filter states:

```
H = dsp.FIRDecimator;
x = cos(pi/8*[0:1023]')+sin(pi/4*[0:1023]');
y = step(H,x);
% Invoke step method again without reset
y1 = step(H,x);
isequal(y,y1) % Returns 0
% Reset filter states
reset(H);
y2 = step(H,x);
isequal(y,y2) % Returns 1
```

Purpose   Decimate input by integer factor

Syntax   Y = step(H,X)

Description   Y = step(H,X) outputs the filtered and downsampled values, Y, of the
input signal, X. A Ki-by-N input matrix is treated as N independent
channels. The length of each column must be a multiple of the
DecimationFactor property value. The FIR decimator operates on
each channel separately and generates a Ko-by-N output matrix where
$Ko = Ki/M$ with M the decimation factor. The object supports real
and complex floating-point and fixed-point inputs, except for complex
unsigned fixed-point inputs.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

Purpose   Static or time-varying FIR filter

Description   The FIRFilter object filters each channel of the input using static or time-varying FIR filter implementations.

To filter each channel of the input:

1 Define and set up your FIR filter. See "Construction" on page 5-473.

2 Call step to filter each channel of the input according to the properties of dsp.FIRFilter. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.FIRFilter returns a default FIR filter object, H, which independently filters each channel of the input over successive calls to the step method, using a specified FIR filter implementation. The default numerator coefficients are [0.5 0.5].

H = dsp.FIRFilter('*PropertyName*',*PropertyValue*, ...) returns an FIR filter system object, H, with each property set to the specified value.

Properties   **Structure**

Filter structure

Specify the filter structure.

You can specify the filter structure as one of | Direct form | Direct form symmetric | Direct form antisymmetric | Direct form transposed | Lattice MA |. The default is Direct form.

NumeratorSource

Source of filter coefficients

Specify the source of the filter coefficients as one of | Property | Input port |. The default is Property. When you specify Input port, the filter object updates the time-varying filter once

every frame, when the `FrameBasedProcessing` property is `true`. When the `FrameBasedProcessing` property is `false`, it updates once every sample.

This applies when you set the `Structure` to `Direct form` | `Direct form symmetric` | `Direct form antisymmetric` | `Direct form transposed`.

ReflectionCoefficientsSource

Source of filter coefficients

Specify the source of the Lattice filter coefficients as one of | `Property` | `Input port` |. The default is `Property`. When you specify `Input port`, the filter object updates the time-varying filter once every frame, when the `FrameBasedProcessing` property is `true`. When the `FrameBasedProcessing` property is `false`, it updates once every sample.

This applies when you set the `Structure` to `Lattice MA`.

Numerator

Numerator coefficients

Specify the filter coefficients as a real or complex numeric row vector. This property applies when you set the `NumeratorSource` property to `Property`, and the `Structure` property is not set to `Direct form`, `Direct form symmetric`, `Direct form antisymmetric`, or `Direct form transposed`. The default is `[0.5 0.5]`. This property is tunable.

ReflectionCoefficients

Reflection coefficients of lattice filter structure

Specify the reflection coefficients of a lattice filter as a real or complex numeric row vector. This property applies when you set the `Structure` property to `Lattice MA`, and the `ReflectionCoefficientsSource` property to `Property`. The default is `[0.5 0.5]`. This property is tunable.

InitialConditions

Initial conditions for the FIR filter

Specify the initial conditions of the filter states. The number of states or delay elements equals the number of reflection coefficients for the lattice structure, or the number of filter coefficients–1 for the other direct form structures.

You can specify the initial conditions as a scalar, vector, or matrix. If you specify a scalar value, the FIR filter object initializes all delay elements in the filter to that value. If you specify a vector whose length equals the number of delay elements in the filter, each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector of initial conditions to each channel of the input signal.

If you specify a vector whose length equals the product of the number of input channels and the number of delay elements in the filter, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel.

If you specify a matrix with the same number of rows as the number of delay elements in the filter, and one column for each channel of the input signal, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. The default is 0.

FrameBasedProcessing

Enable frame-based processing

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. The default is `true`.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`. This property applies only if the object is not in full precision mode.

CoefficientsDataType

Coefficients word and fraction lengths

Specify the coefficients fixed-point data type as one of | `Same word length as input` | `Custom` |. This property applies when you set the `NumeratorSource` property to `Property`. The default is `Same word length as input`.

CustomCoefficientsDataType

Custom coefficients word and fraction lengths

Specify the coefficients fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `CoefficientsDataType` property to `Custom`. The default is `numerictype(true,16,15)`.

ReflectionCoefficientsDataType

Reflection coefficients word and fraction lengths

Specify the reflection coefficients fixed-point data type as one of | `Same word length as input` | `Custom` |. This property applies when you set the `ReflectionCoefficientsSource` property to `Property`. The default is `Same word length as input`.

CustomReflectionCoefficientsDataType

Custom reflection coefficients word and fraction lengths

Specify the numerator fixed-point type as a signed or unsigned `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ReflectionCoefficientsDataType` property to `Custom`. The default is `numerictype(true,16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomProductDataType

Custom product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numerictype(true,32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type to one of | Full precision | Same as input | Same as product | Custom |. The default is Full precision.

CustomAccumulatorDataType

Custom accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the AccumulatorDataType property to Custom. The default is numerictype(true,32,30).

StateDataType

State word and fraction lengths

Specify the state fixed-point data type as one of | Same as input | Same as accumulator | Custom |. This property does not apply to any of the direct form or direct form I filter structures. The default is Same as accumulator.

CustomStateDataType

Custom state word and fraction lengths

Specify the state fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the StateDataType property to Custom. The default is numerictype(true,16,15).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | Same as input | Same as accumulator | Custom |. The default is Same as accumulator.

CustomOutputDataType

Custom output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype(true,16,15)`.

Methods

clone	Create static or time-varying FIR filter with same property values
freqz	Frequency response
fvtool	Open filter visualization tool
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
impz	Impulse response
isLocked	Locked status for input attributes and nontunable properties
phasez	Unwrapped phase response
release	Allow property value and input characteristics changes
reset	Reset internal states of FIR filter
step	Filter input with FIR filter object

More "Analysis Methods for Filter System Objects" on page 4-2.

Examples

Use an FIR filter to apply a low pass filter to a waveform with two sinusoidal components:

```
t = [0:63]./32e3;
xin = (sin(2*pi*4e3*t)+sin(2*pi*12e3*t)) / 2;

hSR = dsp.SignalSource(xin', 4);
hLog = dsp.SignalSink;
```

```
hFIR = dsp.FIRFilter;
hFIR.Numerator = fir1(10,0.5);

while ~isDone(hSR)
     input = step(hSR);
     filteredOutput = step(hFIR,input);
     step(hLog,filteredOutput);
end

filteredResult = hLog.Buffer;
sper = spectrum.periodogram;
psd(sper,filteredResult,'Fs',32e3)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Discrete FIR Filter block reference page. The object properties correspond to the block parameters.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the FrameBasedProcessing property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.BiquadFilter

Purpose   Create static or time-varying FIR filter with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a FIR filter object, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

dsp.FIRFilter.freqz

Purpose   Frequency response

Syntax   ```
                  [h,w] = freqz(H)
                  [h,w] = freqz(H,n)
                  [h,w] = freqz(H,Name,Value)
                  freqz(H)
                  ```

Description   `[h,w] = freqz(H)` returns the complex, 8192–element frequency
                  response vector `h`, and the corresponding frequencies `w` in
                  radians/sample.

                  `[h,w] = freqz(H,n)` returns the complex, n-element frequency
                  response vector `h`, and the corresponding frequencies `w` in
                  radians/sample, using `n` samples.

                  `[h,w] = freqz(H,Name,Value)` returns the frequency response and
                  the corresponding frequencies, with additional options specified by one
                  or more Name, Value pair arguments.

                  `freqz(H)` uses FVTool to plot the magnitude and unwrapped phase of
                  the frequency response of the filter System object `H`.

Purpose   Open filter visualization tool

Syntax   ```
fvtool(H)
fvtool(H,'Arithmetic',ARITH,...)
```

**Description**  `fvtool(H)` performs an analysis and computes the magnitude response of the filter System object `H`.

fvtool(H,'Arithmetic',ARITH,...) analyzes the filter System object H, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

# dsp.FIRFilter.getNumInputs

**Purpose**　　　　Number of expected inputs to step method

**Syntax**　　　　`N = getNumInputs(H)`

**Description**　　`N = getNumInputs(H)` returns the number of expected inputs, `N`, to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

**Purpose**      Number of outputs of step method

**Syntax**       N = getNumOutputs(H)

**Description**   N = getNumOutputs(H) returns the number of outputs, N, of the step
                 method.

                 The getNumOutputs method returns a positive integer that is the
                 number of outputs from the step method. This value will change if you
                 alter any properties that turn outputs on or off.

# dsp.FIRFilter.impz

**Purpose**      Impulse response

**Syntax**
```
[h,t] = impz(H)
[h,t] = impz(H,Name,Value)
impz(H)
```

**Description**    `[h,t] = impz(H)` returns the impulse response `h`, and the corresponding time points `t` at which the impulse response of`H` is computed.

`[h,t] = impz(H,Name,Value)` returns the impulse response `h`, and the corresponding time points`t` , with additional options specified by one or more Name, Value pair arguments.

`impz(H)` uses FVTool to plot the impulse response of the filter System object `H`.

**Note** You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

**Purpose**     Locked status for input attributes and nontunable properties

**Syntax**      isLocked(H)

**Description**  isLocked(H) returns the locked state of the FIR filter.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

# dsp.FIRFilter.phasez

**Purpose**        Unwrapped phase response

**Syntax**
```
[phi,w] = phasez(H)
[phi,w] = phasez(H,n)
[phi,w] = phasez(H,Name,Value)
phasez(H)
```

**Description**     `[phi,w] = phasez(H)` returns the 8192–element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample.

`[phi,w] = phasez(H,n)` returns the n-element phase response vector `phi`, and the corresponding frequencies `w` in radians/sample, using `n` samples.

`[phi,w] = phasez(H,Name,Value)` returns the phase response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`phasez(H)` uses FVTool to plot the phase response of the filter System object `H`.

**Purpose**      Allow property value and input characteristics changes

**Syntax**       `release(H)`

**Description**  `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

---

**Note**  You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

# dsp.FIRFilter.reset

**Purpose**    Reset internal states of FIR filter

**Syntax**    `reset(H)`

**Description**    `reset(H)` resets the filter states of the FIR filter object, `H`, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the `step` method applies the FIR filter object to nonzero input data, the states may be nonzero. Invoking the `step` method again without first invoking the `reset` method may produce different outputs for an identical input.

For example:

```
H = dsp.FIRFilter;
H.Numerator = fir1(20,0.25);
n = 0:100;
x = cos(0.2*pi*n)+sin(0.8*pi*n);
y = step(H,x);
% Filter states are nonzero
% Invoke step method again without resetting states
y1 = step(H,x);
isequal(y,y1) % returns 0
% Now reset filter states to 0
reset(H)
% Invoke step method
y2 = step(H,x);
isequal(y,y2) % returns a 1
```

Purpose   Filter input with FIR filter object

Syntax   Y = step(HFIR,X)
              Y = step(HFIR,X,COEFF)

Description   Y = step(HFIR,X) filters the real or complex input signal X using the
                 FIR filter, H, to produce the output Y. When the input data is of a
                 fixed-point type, it must be signed when the structure is set to Direct
                 form symmetric or Direct form antisymmetric. The FIR filter
                 object operates on each channel of the input signal independently over
                 successive calls to step method.

                 Y = step(HFIR,X,COEFF) uses the time-varying coefficients,
                 COEFF, to filter the input signal X and produce the output Y.
                 You can use this option when you set the NumeratorSource or
                 ReflectionCoefficientsSource property to Input port.

                 Note The object performs an initialization the first time the step
                 method is executed. This initialization locks nontunable properties and
                 input specifications, such as dimensions, complexity, and data type
                 of the input data. If you change a nontunable property or an input
                 specification, the System object issues an error. To change nontunable
                 properties or inputs, you must first call the release method to unlock
                 the object.

                 ---

dsp.FIRInterpolator

Purpose   Polyphase FIR interpolator

Description   The FIRInterpolator object upsamples an input by the integer
upsampling factor, *L*, followed by an FIR anti-imaging filter.
The filter coefficients are scaled by the interpolation factor. A
polyphase interpolation structure implements the filter. The resulting
discrete-time signal has a sampling rate *L* times the original sampling
rate.

To upsample an input:

1 Define and set up your FIR interpolator. See "Construction" on page
5-492.

2 Call step to upsample the input according to the properties of
dsp.FIRInterpolator. The behavior of step is specific to each object
in the toolbox.

Construction   H= dsp.FIRInterpolator returns an FIR interpolator, H, which
upsamples an input signal by a factor of 3 and applies an FIR filter
to interpolate the output.

H = dsp.FIRInterpolator('*PropertyName*',*PropertyValue*, ...)
returns an FIR interpolator, H, with each property set to the specified
value.

H = dsp.FIRInterpolator(INTERP,
NUM,'*PropertyName*',*PropertyValue*, ...) returns an FIR
interpolation object, H, with the InterpolationFactor property set to
INTERP, the Numerator property set to NUM, and other properties set to
the specified values.

Properties   **InterpolationFactor**

Interpolation factor

Specify the integer factor, *L*, by which to increase the sampling
rate of the input signal. The polyphase implementation uses *L*
polyphase subfilters to compute convolutions at the lower sample

rate. The FIR interpolator delays and interleaves these lower-rate convolutions to obtain the higher-rate output. The property value defaults to 3.

Numerator

FIR filter coefficients

Specify the numerator coefficients of the FIR anti-imaging filter as the coefficients of a polynomial in z^{-1}. Indexing from zero, the filter coefficients are:

$$H(z) = \sum_{n=0}^{N-1} b(n)z^{-n}$$

To act as an effective anti-imaging filter, the coefficients must correspond to a lowpass filter with a normalized cutoff frequency no greater than the reciprocal of the InterpolationFactor. The filter coefficients are scaled by the value of the InterpolationFactor property before filtering the signal. To form the *L* polyphase subfilters, Numerator is appended with zeros if necessary. The default is the output of fir1(15,0.25).

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set FullPrecisionOverride to false, fixed-point data types are controlled through individual fixed-point property settings.

For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap. This property applies only if the object is not in full precision mode.

CoefficientsDataType

Coefficient word and fraction lengths

Specify the coefficients fixed-point data type as one of | Same word length as input | Custom |. The default is Same word length as input.

CustomCoefficientsDataType

Coefficient word and fraction lengths

Specify the coefficients fixed-point type as a numerictype object with a Signedness of Auto. This property applies only when the CoefficientsDataType property is Custom. The default is numerictype([],16,15).

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | Full precision | Same as input | Custom |. The default is Full precision.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `ProductDataType` property is `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | `Full precision` | `Same as product` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `AccumulatorDataType` property is `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | `Same as accumulator` | `Same as product` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `OutputDataType` property is `Custom`. The default is `numerictype([],16,15)`.

dsp.FIRInterpolator

Methods

clone	Create FIR interpolator object with same property values
freqz	Frequency response
fvtool	Open filter visualization tool
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
impz	Impulse response
isLocked	Locked status for input attributes and nontunable properties
phasez	Unwrapped phase response
release	Allow property value and input characteristics changes
reset	Reset FIR interpolator filter states
step	Upsample and interpolate input

More "Analysis Methods for Filter System Objects" on page 4-2.

Examples

Double the sampling rate of an audio signal from 22.05 kHz to 44.1 kHz, and play the audio:

```
hmfr = dsp.AudioFileReader('OutputDataType',...
   'single');
hap = dsp.AudioPlayer(44100);
hfirint = dsp.FIRInterpolator(2, ...
   firpm(30, [0 0.45 0.55 1], [1 1 0 0]));

while ~isDone(hmfr)
    frame = step(hmfr);
    y = step(hfirint, frame);
```

```
    step(hap, y);
end

pause(1);
release(hmfr);
release(hap);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the FIR Interpolation block reference page. The object properties correspond to the block parameters, except:

- The FIRInterpolator object does not have a property that corresponds to the **Input processing** parameter of the FIR Interpolation block.

- The **Rate options** block parameter is not supported by the FIRInterpolator object.

See Also   dsp.FIRDecimator | dsp.FIRRateConverter

dsp.FIRInterpolator.clone

Purpose   Create FIR interpolator object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates an FIR interpolator, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Frequency response

Syntax   ```
[h,w] = freqz(H)
[h,w] = freqz(H,n)
[h,w] = freqz(H,Name,Value)
freqz(H)
```

Description   `[h,w] = freqz(H)` returns the complex, 8192–element frequency
                  response vector h, and the corresponding frequencies w in
                  radians/sample.

                  `[h,w] = freqz(H,n)` returns the complex, n-element frequency
                  response vector h, and the corresponding frequencies w in
                  radians/sample, using n samples.

                  `[h,w] = freqz(H,Name,Value)` returns the frequency response and
                  the corresponding frequencies, with additional options specified by one
                  or more Name, Value pair arguments.

                  `freqz(H)` uses FVTool to plot the magnitude and unwrapped phase of
                  the frequency response of the filter System object H.

Purpose	Open filter visualization tool

Syntax

```
fvtool(H)
fvtool(H,'Arithmetic',ARITH,...)
```

Description fvtool(H) performs an analysis and computes the magnitude response of the filter System object H.

fvtool(H,'Arithmetic',ARITH,...) analyzes the filter System object H, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.FIRInterpolator.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of output from the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Impulse response

Syntax   [h,t] = impz(H)
                [h,t] = impz(H,Name,Value)
                impz(H)

Description [h,t] = impz(H) returns the impulse response h, and the
                corresponding time points t at which the impulse response ofH is
                computed.

                [h,t] = impz(H,Name,Value) returns the impulse response h, and the
                corresponding time pointst , with additional options specified by one or
                more Name, Value pair arguments.

                impz(H) uses FVTool to plot the impulse response of the filter System
                object H.

                Note You can use impz for both real and complex filters. When you
                omit the output arguments, impz plots only the real part of the impulse
                response.

                ---

dsp.FIRInterpolator.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   `isLocked(H)`

Description   `isLocked(H)` returns the locked state of the FIR interpolator.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Unwrapped phase response

Syntax   ```
                   [phi,w] = phasez(H)
                   [phi,w] = phasez(H,n)
                   [phi,w] = phasez(H,Name,Value)
                   phasez(H)
                   ```

**Description**    `[phi,w] = phasez(H)` returns the 8192–element phase response vector
                   `phi`, and the corresponding frequencies `w` in radians/sample.

                   `[phi,w] = phasez(H,n)` returns the n-element phase response vector
                   `phi`, and the corresponding frequencies `w` in radians/sample, using
                   `n` samples.

                   `[phi,w] = phasez(H,Name,Value)` returns the phase response and the
                   corresponding frequencies, with additional options specified by one or
                   more Name, Value pair arguments.

                   `phasez(H)` uses FVTool to plot the phase response of the filter System
                   object `H`.

# dsp.FIRInterpolator.release

**Purpose**        Allow property value and input characteristics changes

**Syntax**         `release(H)`

**Description**    `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

---

**Note**  You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

**Purpose**      Reset FIR interpolator filter states

**Syntax**       reset(H)

**Description**  reset(H) resets the filter states of the FIR filter in the interpolator object, H, to their initial values of 0. The initial filter state values correspond to the initial conditions for the constant coefficient linear difference equation defining the FIR filter. After the step method applies the interpolator to nonzero input data, the states may be nonzero. Invoking the step method again without first invoking the reset method may produce different outputs for an identical input.

Reset filter states to 0 to produce consistent output:

```
H = dsp.FIRInterpolator(2);
x =[1 -1]'; x = repmat(x,8,1);
y = step(H,x); % Filter states are nonzero
% Use reset method to set states to zero
reset(H);
% Apply FIR interpolator to input x
y1 = step(H,x);
isequal(y,y1)
% Returns a 1
```

Purpose   Upsample and interpolate input

Syntax   Y = step(H,X)

Description   Y = step(H,X) outputs the upsampled and interpolated values, Y , of
                 the input signal X . A *K*-by-*N* input matrix is treated as *N* independent
                 channels. The FIR interpolator object interpolates each channel over
                 the first dimension and generates a *M*-by-*N* output matrix, where *M* is
                 the product of *K* and the upsampling factor, *L*.

> **Note** The object performs an initialization the first time the step
> method is executed. This initialization locks nontunable properties and
> input specifications, such as dimensions, complexity, and data type
> of the input data. If you change a nontunable property or an input
> specification, the System object issues an error. To change nontunable
> properties or inputs, you must first call the release method to unlock
> the object.

Purpose   Sample rate converter

Description   The FIRRateConverter performs sampling rate conversion by a rational factor on a vector or matrix input. The FIR rate convertor cascades an interpolator with a decimator. The interpolator upsamples the input by the upsampling factor, *L*, followed by a lowpass FIR filter. The FIR filter acts both as an anti-imaging filter and an anti-aliasing filter prior to decimation. The decimator downsamples the output of upsampling and FIR filtering by the downsampling factor *M*. You must use upsampling and downsampling factors that are relatively prime, or coprime. The resulting discrete-time signal has a sampling rate *L/M* times the original sampling rate.

To perform sampling rate conversion:

1 Define and set up your FIR sample rate converter. See "Construction" on page 5-509.

2 Call step to perform sampling rate conversion according to the properties of dsp.FIRRateConverter. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.FIRRateConverter returns a FIR sample rate converter, H, that resamples an input signal at a rate 3/2 times the original sampling rate.

H = dsp.FIRRateConverter('*PropertyName*',*PropertyValue*, ...) returns an FIR sample rate converter, H, with each property set to the specified value.

H = dsp.FIRRateConverter(L,M,NUM,'*PropertyName*', *PropertyValue*, ...) returns an FIR sample rate converter, H, with the InterpolationFactor property set to L, the DecimationFactor property set to M, the Numerator property set to NUM, and other specified properties set to the specified values.

Properties   **InterpolationFactor**

Interpolation factor

Specify the integer upsampling factor. The default is 3.

DecimationFactor

Decimation factor

Specify the integer downsampling factor. The default is 2.

Numerator

FIR filter coefficients

Specify the FIR filter coefficients in powers of z^{-1}. The length of filter coefficients must exceed the interpolation factor. Use a lowpass with normalized cutoff frequency no greater than `min(1/InterpolationFactor, 1/DecimationFactor)`. All initial filter states are zero. The default is `firpm(70,[0 0.28 0.32 1],[1 1 0 0])`.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor. This property applies only if the object is not in full precision mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap. This property applies only if the object is not in full precision mode.

CoefficientsDataType

Word and fraction lengths of filter coefficients

Specify the filter coefficient fixed-point data type as one of | Same word length as input | Custom |. The default is Same word length as input.

CustomCoefficientsDataType

Word and fraction lengths of filter coefficients

Specify the filter coefficient fixed-point type as a numerictype object with a Signedness of Auto. This property applies only when the CoefficientsDataType property is Custom. The default is numerictype([],16,15).

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | Full precision | Same as input | Custom |. The default is Full precision.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only

when the `ProductDataType` property is `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | `Full precision` | `Same as product` | `Same as input` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `AccumulatorDataType` property is `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | `Same as accumulator` | `Same as product` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `OutputDataType` property is `Custom`. The default is `numerictype([],16,15)`.

Methods		
	clone	Create FIR sample rate convertor object with same property values
	getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of FIR sample rate converter
step	Resample input with FIR rate converter

Examples   Resample a 100 Hz sine wave signal by a factor of 3:2:

```
hsin = dsp.SineWave(1, 100,'SampleRate', 5000,'SamplesPerFrame', 50);

% Create a FIR rate converter filter. The default interpolation
% factor is 3 and decimation factor is 2.
hfirrc = dsp.FIRRateConverter;
input = step(hsin);
output = step(hfirrc, input);

% Plot the original and resampled signals.
ndelay = round(length(hfirrc.Numerator)/2/hfirrc.DecimationFactor);
indx = ndelay+1:length(output);
x = (0:length(indx)-1)/hsin.SampleRate*hfirrc.DecimationFactor/hfirrc.
stem((0:38)/hsin.SampleRate, input(1:39)); hold on;
stem(x, hfirrc.InterpolationFactor*output(indx),'r');
legend('Original','Resampled');
```

Resample and play an audio signal from 48 kHz to 32 kHz (Windows only):

```
if ispc %This example runs only on Windows
    hmfr = dsp.AudioFileReader('audio48kHz.wav', ...
```

```
               'OutputDataType', 'single', ...
               'SamplesPerFrame', 300);
             hap = dsp.AudioPlayer(32000);

             % Create an FIRRateConverter System object with interpolation
             % factor = 2, decimation factor = 3. Default FIR filter
             % coefficients define a lowpass filter with normalized
             % cutoff frequency of 1/3.
             hfirrc = dsp.FIRRateConverter(2,3);

             while ~isDone(hmfr)
               audio1 = step(hmfr);
               audio2 = step(hfirrc, audio1);
               step(hap, audio2);
             end

             release(hmfr);
             release(hap);

         else
             disp('This example runs only on Windows platforms.');
         end
```

Algorithms   This object implements the algorithm, inputs, and outputs described on
the FIR Rate Conversion block reference page. The object properties
correspond to the block parameters.

See Also   dsp.FIRInterpolator | dsp.FIRDecimator

Purpose   Create FIR sample rate convertor object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an FIR sample rate converter object, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

dsp.FIRRateConverter.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
method.

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the FIR sample rate converter.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.FIRRateConverter.reset

Purpose   Reset states of FIR sample rate converter

Syntax   reset(H)

Description reset(H) resets the filter states of the FIR filter in the sample rate converter, H, to their initial values of 0. The initial filter state values correspond to the initial conditions for the constant coefficient linear difference equation defining the FIR filter. After the step method applies the FIR rate converter to nonzero input data, the states may be nonzero. Invoking the step method again without first invoking the reset method may produce different outputs for an identical input.

Reset filter states to 0 to produce consistent output:

```
H = dsp.FIRRateConverter(2,4);
x =[1 -1]'; x = repmat(x,8,1);
y = step(H,x); % Filter states are nonzero
% Use reset method to set states to zero
reset(H);
% Apply sampling rate converter to input x
y1 = step(H,x);
isequal(y,y1)
% Returns a 1
```

Purpose   Resample input with FIR rate converter

Syntax   Y = step(H,X)

Description   Y = step(H,X) resamples the input X and returns the resampled signal
                   Y . An *M*-by-*N* matrix input is treated as *N* independent channels.

dsp.Histogram

Purpose	Histogram of input or sequence of inputs

Description   The Histogram object generates a histogram for an input or a sequence of inputs.

To generate a histogram for an input or a sequence of inputs:

1 Define and set up your histogram object. See "Construction" on page 5-522.

2 Call step to generate the histogram for an input according to the properties of dsp.Histogram. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.Histogram returns a histogram object, H, that computes the frequency distribution of the elements in each input matrix.

H = dsp.Histogram(*'PropertyName'*,*PropertyValue*, ...) returns a histogram object, H, with each specified property set to the specified value.

H = dsp.Histogram(*MIN*,*MAX*,*NUMBINS*,*'PropertyName'*,*PropertyValue*,...) returns a histogram object, H, with the LowerLimit property set to *MIN*, UpperLimit property set to *MAX*, NumBins property set to *NUMBINS* and other specified properties set to the specified values.

Properties   **LowerLimit**

Lower boundary

Specify the lower boundary of the lowest-valued bin as a real-valued scalar. NaN and Inf are not valid values for this property. The default is 0. This property is tunable.

UpperLimit

Upper boundary

Specify the upper boundary of the highest-valued bin as a real-valued scalar. NaN and Inf are not valid values for this property. The default is 10. This property is tunable.

NumBins

Number of bins in histogram

Specify the number of bins in the histogram. The default is 11.

Dimension

Specify how the histogram calculation is performed over the data as one of | All | Column |. The default is Column.

Normalize

Enable output vector normalization

Specify whether the histogram object normalizes the output vector, v, so that $sum(v) = 1$. When you set this property to true, the output vector is normalized. When you set it to false, the object supports fixed-point operations and does not use this property for normalization. The default is false.

RunningHistogram

Enable calculation over successive step method calls

Set this property to true to enable running histogram calculations for the input elements over successive calls to the step method. Set this property to false to compute a histogram for the current input. The default is false.

ResetInputPort

Enable resetting in running histogram mode

Set this property to true to enable resetting the running histogram. When you set the property to true, specify a reset input to the step method that resets the running histogram. This property applies when you set the RunningHistogram property to true. When this property is false, the histogram object does not reset. The default is false.

ResetCondition

Reset condition for running histogram mode

Specify the event that resets the running histogram as one of | Rising edge | Falling edge | Either edge | Non-zero |. This property applies when you set the ResetInputPort property to true. The default is Non-zero

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomProductDataType

Custom product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the ProductDataType property to Custom. The default is numerictype([],32,30).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | Same as product | Same as input | Custom |. The default is Same as input.

CustomAccumulatorDataType

Custom accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the AccumulatorDataType property to Custom. The default is numerictype([],32,30).

Methods

clone	Create histogram object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset histogram bin values to zero
step	Histogram for input data

Examples

Compute a histogram with four bins, for possible input values 1 through 4:

```
hhist = dsp.Histogram(1,4,4);
y = step(hhist, [1 2 2 3 3 3 4 4 4 4]');
% y equals [1; 2; 3; 4] - one one, two twos, etc.
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Histogram block reference page. The object properties correspond to the block parameters, except:

- The **Reset port** block parameter corresponds to both the ResetCondition and the ResetInputPort object properties.

- The **Find histogram over** block parameter corresponds to the Dimension property of the object.

See Also   dsp.Maximum | dsp.Minimum | dsp.Mean

Purpose   Create histogram object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a Histogram System object C, with the same
property values as H. The clone method creates a new unlocked object
with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                 method.

                 The getNumOutputs method returns a positive integer that is the
                 number of outputs from the step method. This value will change if you
                 alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   `isLocked(H)`

Description   `isLocked(H)` returns the locked state of the Histogram System object.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

> **Note** You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

dsp.Histogram.reset

Purpose   Reset histogram bin values to zero

Syntax   reset(H)

Description   reset(H) sets the Histogram object bin values to zero when you set the RunningHistogram property to true.

Purpose   Histogram for input data

Syntax   Y = step(H,X)
Y = step(H,X,R)

Description   Y = step(H,X) returns a histogram Y for the input data X . When the
RunningHistogram property is true, Y corresponds to the histogram of
the input elements over successive calls to the step method.

Y = step(H,X,R) resets the histogram state based on the value of R
and the object's ResetCondition property. You can reset the histogram
state only when the RunningHistogram and the ResetInputPort
properties are true.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

Purpose	Inverse discrete cosine transform (IDCT)
Description	The IDCT object computes the inverse discrete cosine transform (IDCT) of an input.

To compute the IDCT of an input:

1 Define and set up your IDCT object. See "Construction" on page 5-534.

2 Call step to compute the IDCT of an input according to the properties of dsp.IDCT. The behavior of step is specific to each object in the toolbox.

Construction

H = dsp.IDCT returns a inverse discrete cosine transform (IDCT) object, H. This object computes the IDCT of a real or complex input signal using the Table lookup method.

H = dsp.IDCT('*PropertyName*',*PropertyValue*,...) returns an inverse discrete cosine transform (IDCT) object, H, with each property set to the specified value.

Properties

SineComputation

Method to compute sines and cosines

Specify how the IDCT object computes the trigonometric function values as Trigonometric function or Table lookup. You must set this property to Table lookup for fixed-point inputs. The default is Table lookup.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of Ceiling, Convergent, Floor , Nearest, Round, Simplest, or Zero. This property applies when you set the SineComputation property to Table lookup.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of Wrap or Saturate. This property applies when you set the SineComputation property to Table lookup.

SineTableDataType

Sine table word-length designation

Specify the sine table fixed-point data type as one of Same word length as input or Custom. This property applies when you set the SineComputation property to Table lookup.

CustomSineTableDataType

Sine table word length

Specify the sine table fixed-point type as a signed, unscaled numerictype object. This property applies when you set the SineComputation property to Table lookup and the SineTableDataType property to Custom. The default is numerictype(true,16).

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of Full precision, Same as input, Custom. This property applies when you set the SineComputation property to Table lookup.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a signed, scaled numerictype object. This property applies when you set the SineComputation property to Table lookup and the ProductDataType property to Custom. The default is numerictype(true,32,30).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Full precision`, `Same as input`, `Same as product`, or `Custom`. This property applies when you set the `SineComputation` property to `Table lookup`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a signed, scaled `numerictype` object. This property applies when you set the `SineComputation` property to `Table lookup` and the `AccumulatorDataType` property to `Custom`. The default is `numerictype(true,32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of `Full precision`, `Same as input`, `Custom`. This property applies when you set the `SineComputation` property to `Table lookup`. The default is `Full precision`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a signed, scaled `numerictype` object. This property applies when you set the `SineComputation` property to `Table lookup` and the `OutputDataType` property to `Custom`. The default is `numerictype(true,16,15)`.

Methods

clone	Create inverse discrete cosine transform object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Inverse discrete cosine transform (IDCT) of input

Examples

Use DCT to analyze the energy content in a sequence:

```
x = (1:128).' + 50*cos((1:128).'*2*pi/40);
hdct = dsp.DCT;

X = step(hdct, x);
% Set the DCT coefficients which represent less
% than 0.1% of the total energy to 0 and
% reconstruct the sequence using IDCT.
[XX, ind] = sort(abs(X),1,'descend');
ii = 1;

while (norm([XX(1:ii);zeros(128-ii,1)]) <= 0.999*norm(XX))
    ii = ii+1;
end

disp(['Number of DCT coefficients that represent 99.9%',...
'of the total energy in the sequence: ',num2str(ii)]);
XXt = zeros(128,1);
XXt(ind(1:ii)) = X(ind(1:ii));
```

```
hidct = dsp.IDCT;
xt = step(hidct, XXt);

plot(1:128,[x xt]);
legend('Original signal','Reconstructed signal',...
'location','best');
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the IDCT block reference page. The object properties correspond to the block parameters.

See Also   dsp.DCT | dsp.FFT | dsp.IFFT.

Purpose   Create inverse discrete cosine transform object with same property
                  values

Syntax   C = clone(H)

Description   C = clone(H) creates a copy of the current IDCT object, C, with the
                  same property values as H. The clone method creates a new unlocked
                  object.

dsp.IDCT.getNumInputs

Purpose　　　　Number of expected inputs to step method

Syntax　　　　`N = getNumInputs(H)`

Description　　`N = getNumInputs(H)` returns the number of expected inputs, N, to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of step method

Syntax   `N = getNumOutputs(H)`

Description   `N = getNumOutputs(H)` returns the number of outputs, `N`, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

dsp.IDCT.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the IDCT object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Inverse discrete cosine transform (IDCT) of input

Syntax   Y = step(H,X)

Description   Y = step(H,X) computes the inverse discrete cosine transform, Y , of input X.

> **Note** The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Inverse discrete Fourier transform (IDFFT)

Description   The IFFT object computes the inverse discrete Fourier transform
(IDFFT) of the input. The object uses one or more of the following fast
Fourier transform (FFT) algorithms depending on the complexity of the
input and whether the output is in linear or bit-reversed order:

- Double-signal algorithm

- Half-length algorithm

- Radix-2 decimation-in-time (DIT) algorithm

- Radix-2 decimation-in-frequency (DIF) algorithm

- An algorithm chosen by FFTW [1], [2]

To compute the IFFT of the input:

1 Define and set up your IFFT object. See "Construction" on page 5-545.

2 Call step to compute the IFFT of the input according to the
properties of dsp.IFFT. The behavior of step is specific to each object
in the toolbox.

Construction   H = dsp.IFFT returns an IFFT object, H, that computes the IDFT of a
column vector or *N*-D array. For column vectors or N-D arrays, the IFFT
object computes the IDFT along the first dimension of the array. If the
input is a row vector, the IFFT object computes a row of single-sample
IDFTs and issues a warning.

H = dsp.IFFT('*PropertyName*',*PropertyValue*, ...) returns an
IFFT object, H, with each property set to the specified value.

Properties   **FFTImplementation**

FFT implementation

Specify the implementation used for the FFT as one of Auto |
Radix-2 | FFTW. When you set this property to Radix-2, the FFT
length must be a power of two.

BitReversedInput

Enable bit-reversed order interpretation of input elements

Set this property to true if the order of Fourier transformed
input elements to the IFFT object are in bit-reversed order. This
property applies only when the FFTLengthSource property is
Auto. The default is false, which denotes linear ordering.

ConjugateSymmetricInput

Enable conjugate symmetric interpretation of input

Set this property to true if the input is conjugate symmetric to
yield real-valued outputs. The discrete Fourier transform of a real
valued sequence is conjugate symmetric, and setting this property
to true optimizes the IDFT computation method. Setting this
property to false for conjugate symmetric inputs may result
in complex output values with nonzero imaginary parts. This
occurs due to rounding errors. Setting this property to true
for nonconjugate symmetric inputs results in invalid outputs.
This property applies only when the FFTLengthSource property
is Auto. The default is false.

Normalize

Enable dividing output by FFT length

Specify whether to divide the IFFT output by the FFT length.
The default is true and each element of the output is divided
by the FFT length.

FFTLengthSource

Source of FFT length

Specify how to determine the FFT length as Auto or Property.
When you set this property to Auto, the FFT length equals the
number of rows of the input signal. This property applies only

when both the `BitReversedInput` and `ConjugateSymmetricInput` properties are `false`. The default is `Auto`.

FFTLength

FFT length

Specify the FFT length as a numeric scalar. This property applies when you set the `BitReversedInput` and `ConjugateSymmetricInput` properties to `false`, and the `FFTLengthSource` property to `Property`. The default is `64`.

This property must be a power of two when the input is a fixed-point data type, or when you set the `FFTImplementation` property to `Radix-2`.

When you set the `FFT implementation` property to `Radix-2`, or when you set the `BitReversedOutput` property to `true`, this value must be a power of two.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

SineTableDataType

Sine table word and fraction lengths

Specify the sine table data type as `Same word length as input` or `Custom`. The default is `Same word length as input`.

CustomSineTableDataType

Sine table word and fraction lengths

Specify the sine table fixed-point type as an unscaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `SineTableDataType` property to `Custom`. The default is `numerictype([],16)`.

ProductDataType

Product word and fraction lengths

Specify the product data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator data type as `Full precision`, `Same as input`, `Same as product`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

Methods

clone	Create IFFT object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Inverse discrete Fourier transform of input

Examples

Use DFT and IDFT to analyze the energy content in a sequence:

```
Fs = 40; L = 128;
t = (0:L-1)'/Fs;
x = sin(2*pi*10*t) + 0.75*cos(2*pi*15*t);
y = x + .5*randn(size(x)); % noisy signal
hfft = dsp.FFT;
hifft = dsp.IFFT('ConjugateSymmetricInput', true);
X = step(hfft, x);
[XX, ind] = sort(abs(X),1,'descend');
XXn = sqrt(cumsum(XX.^2))/norm(XX);
```

```
ii = find(XXn > 0.999, 1);
disp('Number of FFT coefficients that represent 99.9% ')
disp(['of the total energy in the sequence: ', num2str(ii)]);
XXt = zeros(128,1);
XXt(ind(1:ii)) = X(ind(1:ii));
xt = step(hifft, XXt);
% Verify the reconstructed signal matches the original
norm(x-xt)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the IFFT block reference page. The object properties correspond to the block parameters, except:

> **Output sampling mode** parameter is not supported by dsp.IFFT.

References   [1] FFTW (http://www.fftw.org)

[2] Frigo, M. and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT,"*Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Vol. 3, 1998, pp. 1381-1384.

See Also   dsp.FFT | dsp.DCT | dsp.IDCT

Purpose   Create IFFT object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an IFFT object, C, with the same property values
                  as H. The clone method creates a new unlocked object.

dsp.IFFT.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

dsp.IFFT.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the IFFT object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Inverse discrete Fourier transform of input

Syntax   `Y = step(H,X)`

Description   `Y = step(H,X)` computes the inverse discrete Fourier transform
(IDFT) , `Y` , of the input `X` along the first dimension of `X`. When
the `FFTLengthSource` property is `Auto`, the length of `X` along the
first dimension must be a positive integer power of two. When the
`FFTLengthSource` property is `'Property'`, the length of `X` along the
first dimension can be any positive integer and the `FFTLength` property
must be a positive integer power of two.

Note The object performs an initialization the first time the `step`
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the `release` method to unlock
the object.

Purpose   Infinite Impulse Response (IIR) filter

Description   The IIRFilter object filters each channel of the input using IIR filter implementations.

To filter each channel of the input:

1 Define and set up your IIR filter. See "Construction" on page 5-557.

2 Call step to filter each channel of the input according to the properties of dsp.IIRFilter. The behavior of step is specific to each object in the toolbox.

Construction   HIIR = dsp.IIRFilter returns an IIR filter System object, HIIR, which independently filters each channel of the input over successive calls to the step method. This System object uses a specified IIR filter implementation.

HIIR = dsp.IIRFilter('*PropertyName*',*PropertyValue*, ...) returns an IIR filter System object, HIIR, with each property set to the specified value.

Properties   **Structure**

Filter structure

Specify the filter structure.

You can specify the filter structure as one of | Direct form I | Direct form I transposed | Direct form II | Direct form II transposed. The default is Direct form II transposed. This property is nontunable.

Numerator

Filter numerator coefficients

Specify the numerator coefficients as a real or complex numeric row vector. The default is [1 1]. This property is tunable.

Denominator

Filter denominator coefficients

Specify the denominator coefficients as a real or complex numeric row vector. The default value of this property is [1 0.1]. This property is tunable.

InitialConditions

Initial conditions for the IIR filter

Specify the initial conditions of the filter states. This property is applicable when the structure is one of Direct form II | Direct form II transposed. The default value is 0. The number of states equals the max(N,M)-1, where N and M are the number of poles and zeros respectively.

You can specify the initial conditions as a scalar, vector, or matrix. If you specify a scalar value, this System object initializes all delay elements in the filter to that value. You can also specify a vector whose length equals the number of delay elements in the filter. When you do so, each vector element specifies a unique initial condition for the corresponding delay element. The object applies the same vector of initial conditions to each channel of the input signal.

You can also specify a matrix with the same number of rows as the number of delay elements in the filter and one column for each channel of the input signal. In this case, each element specifies a unique initial condition for the corresponding delay element in the corresponding channel. This property is tunable.

NumeratorInitialConditions

Initial conditions on zeros side

Specify the initial conditions of the filter states on the side of the filter structure with the zeros. This property is applicable when the Structure property is one of Direct form I | Direct form I transposed. The default value of this property is 0. The number

of states equals the `max(N,M)-1`, where `N` and `M` are the number of poles and zeros respectively.

You can specify the initial conditions as a scalar, vector, or matrix. When you specify a scalar, the IIR filter initializes all delay elements on the zeros side in the filter to that value. You can specify a vector of length equal to the number of delay elements on the zeros side in the filter. When you do so, each vector element specifies a unique initial condition for the corresponding delay element on the zeros side.

You can also specify a matrix with the same number of rows as the number of delay elements on the zeros side in the filter, and one column for each channel of the input signal. In this case, each element specifies a unique initial condition for the corresponding delay element on the zeros side in the corresponding channel. This property is tunable.

DenominatorInitialConditions

Initial conditions on poles side

Specify the initial conditions of the filter states on the side of the filter structure with the poles. This property is applicable when the Structure property is one of `Direct form I` | `Direct form I transposed`. The default value of this property is `0`. The number of states equals the `max(N,M)-1`, where `N` and `M` are the number of poles and zeros respectively.

You can specify the initial conditions as a scalar, vector, or matrix. When you specify a scalar, the IIR filter initializes all delay elements on the poles side of the filter to that value. You can specify a vector of length equal to the number of delay elements on the poles side in the filter. When you do so, each vector element specifies a unique initial condition for the corresponding delay element on the poles side.

You can also specify a matrix with the same number of rows as the number of delay elements on the poles side in the filter, and one column for each channel of the input signal. In this case, each

element specifies a unique initial condition for the corresponding delay element on the poles side in the corresponding channel. This property is tunable.

FrameBasedProcessing

Process input as frames or as samples

Set this property to `true` to enable frame-based processing. When this property is `true`, the IIR filter treats each column as an independent channel. Set this property to `false` to enable sample-based processing. When this property is `false`, the IIR filter treats each element of the input as an individual channel. The default is `true`. This property is nontunable.

NumeratorCoefficientsDataType

Numerator coefficients word- and fraction-length designations

Specify the numerator coefficients fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property is nontunable.

DenominatorCoefficientsDataType

Denominator coefficients word- and fraction-length designations

Specify the denominator coefficients fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property is nontunable.

Fixed-Point Properties

NumeratorProductDataType

Product word- and fraction-length designations on zeros side

Specify the numerator product fixed-point data type as one of | `Full precision` | `Same as input` | `Custom` |. The default is `Full precision`. This property is nontunable.

DenominatorProductDataType

Product word- and fraction-length designations on poles side

Specify the denominator product fixed-point data type as one of | Full precision | Same as input | Custom |. The default is Full precision. This property is nontunable.

NumeratorAccumulatorDataType

Accumulator word- and fraction-length designations on zeros side

Specify the numerator accumulator fixed-point data type to one of | Full precision | Same as input | Same as product | Custom |. The default is Full precision. This property is nontunable.

DenominatorAccumulatorDataType

Accumulator word- and fraction-length designations on poles side

Specify the denominator accumulator fixed-point data type to one of | Full precision | Same as input | Same as product | Custom |. The default is Full precision. This property is nontunable.

OutputDataType

Output word- and fraction-length designations

Specify the output fixed-point data type as one of | Full precision | Same as input | Custom |. The default is Same as input. This property is nontunable.

StateDataType

State word- and fraction-length designations

Specify the state fixed-point data type as one of | Same as input | Custom |. This property does not apply to Direct form I filter structure. The default is Same as input. This property is nontunable.

MultiplicandDataType

Multiplicand word- and fraction-length designations

Specify the multiplicand fixed-point data type as one of | `Same as input` | `Custom` |. This property is applicable only when the `Structure` property is `Direct form I transposed`. The default is `Same as input`. This property is nontunable.

CustomNumeratorCoefficientsDataType

Numerator coefficients word- and fraction-lengths

Specify the numerator coefficients fixed-point type as an autosigned numerictype object. This property is applicable when the `NumeratorCoefficientsDataType` property is `Custom`. The default value of this property is numerictype ([],16,15). This property is nontunable.

CustomDenominatorCoefficientsDataType

Denominator coefficients word- and fraction-lengths

Specify the denominator coefficients fixed-point type as an autosigned numerictype object. This property is applicable when the `DenominatorCoefficientsDataType` property is `Custom`. The default value of this property is numerictype([], 16, 15) . This property is nontunable.

CustomNumeratorProductDataType

Custom product word- and fraction-lengths on zeros side

Specify the numerator product fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `NumeratorProductDataType` property to `Custom`. The default is numerictype ([],32,30). This property is nontunable.

CustomDenominatorProductDataType

Custom product word- and fraction-lengths on poles side

Specify the denominator product fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `DenominatorProductDataType` property to `Custom`. The default is numerictype ([],32,30). This property is nontunable.

CustomNumeratorAccumulatorDataType

Custom accumulator word- and fraction-lengths on zeros side

Specify the numerator accumulator fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `NumeratorAccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`. This property is nontunable.

CustomDenominatorAccumulatorDataType

Custom accumulator word- and fraction-lengths on poles side

Specify the denominator accumulator fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `DenominatorAccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`. This property is nontunable.

CustomStateDataType

Custom state word- and fraction-lengths

Specify the state fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `StateDataType` property to `Custom`. The default is `numerictype([],16,15)`. This property is nontunable.

CustomOutputDataType

Custom output word- and fraction-lengths

Specify the output fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`. This property is nontunable.

CustomMultiplicandDataType

Custom multiplicand word- and fraction-lengths

Specify the multiplicand fixed-point type as an autosigned scaled `numerictype` object. This property applies when you set

the `MultiplicandDataType` property to `Custom`. The default is `numerictype([],16,15)`. This property is nontunable.

Methods

clone	Create IIR filter with same property values
freqz	Frequency response
fvtool	Open filter visualization tool
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
impz	Impulse response
isLocked	Locked status for input attributes and nontunable properties
phasez	Unwrapped phase response
release	Allow property value and input characteristics changes
reset	Reset internal states of IIR filter
step	Filter input with IIR filter object

More "Analysis Methods for Filter System Objects" on page 4-2.

Examples

Use an IIR filter to apply a lowpass filter to a waveform with two sinusoidal frequencies.

```
t = [0:63]./32e3;
xin = (sin(2*pi*4e3*t)+sin(2*pi*12e3*t)) / 2;

hSR = dsp.SignalSource(xin', 4);
hLog = dsp.SignalSink;
hIIR = dsp.IIRFilter;
[b,a] = butter(6,0.5);
```

```
hIIR.Numerator = b;
hIIR.Denominator = a;

while ~isDone(hSR)
    input = step(hSR);
    filteredOutput = step(hIIR,input);
    step(hLog,filteredOutput);
end

filteredResult = hLog.Buffer;
sper = spectrum.periodogram;
psd(sper,filteredResult,'Fs',32e3)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Discrete Filter block reference page. The object properties correspond to the block parameters.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the FrameBasedProcessing property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.BiquadFilter | dsp.FIRFilter | dsp.AllpoleFilter

dsp.IIRFilter.clone

Purpose   Create IIR filter with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a IIR filter object, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Frequency response

Syntax   ```
[h,w] = freqz(H)
[h,w] = freqz(H,n)
[h,w] = freqz(H,Name,Value)
freqz(H)
```

Description   `[h,w] = freqz(H)` returns the complex, 8192–element frequency response vector h, and the corresponding frequencies w in radians/sample.

`[h,w] = freqz(H,n)` returns the complex, n-element frequency response vector h, and the corresponding frequencies w in radians/sample, using n samples.

`[h,w] = freqz(H,Name,Value)` returns the frequency response and the corresponding frequencies, with additional options specified by one or more Name, Value pair arguments.

`freqz(H)` uses FVTool to plot the magnitude and unwrapped phase of the frequency response of the filter System object H.

dsp.IIRFilter.fvtool

Purpose   Open filter visualization tool

Syntax
```
fvtool(H)
fvtool(H,'Arithmetic',ARITH,...)
```

Description   `fvtool(H)` performs an analysis and computes the magnitude response of the filter System object `H`.

`fvtool(H,'Arithmetic',ARITH,...)` analyzes the filter System object `H`, based on the arithmetic specified in the ARITH input. ARITH can be set to one of 'double', 'single', or 'fixed'. The analysis tool assumes a double precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state. The 'Arithmetic' input is only relevant for the analysis of filter System objects.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.IIRFilter.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Impulse response

Syntax   ```
[h,t] = impz(H)
[h,t] = impz(H,Name,Value)
impz(H)
```

**Description**   `[h,t] = impz(H)` returns the impulse response h, and the
corresponding time points t at which the impulse response of H is
computed.

`[h,t] = impz(H,Name,Value)` returns the impulse response h, and the
corresponding time points t , with additional options specified by one or
more Name, Value pair arguments.

`impz(H)` uses FVTool to plot the impulse response of the filter System
object H.

---

**Note**  You can use impz for both real and complex filters. When you
omit the output arguments, impz plots only the real part of the impulse
response.

---

# dsp.IIRFilter.isLocked

**Purpose**          Locked status for input attributes and nontunable properties

**Syntax**           isLocked(H)

**Description**      isLocked(H) returns the locked state of the IIR filter.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

**Purpose**     Unwrapped phase response

**Syntax**      [phi,w] = phasez(H)
                [phi,w] = phasez(H,n)
                [phi,w] = phasez(H,Name,Value)
                phasez(H)

**Description**   [phi,w] = phasez(H) returns the 8192–element phase response vector
                phi, and the corresponding frequencies w in radians/sample.

                [phi,w] = phasez(H,n) returns the n-element phase response vector
                phi, and the corresponding frequencies w in radians/sample, using
                n samples.

                [phi,w] = phasez(H,Name,Value) returns the phase response and the
                corresponding frequencies, with additional options specified by one or
                more Name, Value pair arguments.

                phasez(H) uses FVTool to plot the phase response of the filter System
                object H.

# dsp.IIRFilter.release

**Purpose**          Allow property value and input characteristics changes

**Syntax**           release(H)

**Description**      release(H) releases system resources, such as memory, file handles,
                     and hardware connections, and lets you change any properties or input
                     characteristics.

---

**Note**  You can use the release method on a System object in code
generated from MATLAB, but once you release its resources, you cannot
use that System object again.

---

**Purpose**       Reset internal states of IIR filter

**Syntax**        reset(H)

**Description**   reset(H) resets the filter states of the IIR filter object, H, to their initial values of 0. The initial filter state values correspond to the initial conditions for the difference equation defining the filter. After the step method applies the IIR filter object to nonzero input data, the states may be nonzero. Invoking the step method again without first invoking the reset method may produce different outputs for an identical input.

For example:

```
H = dsp.IIRFilter;
[b,a] = ellip(20, 0.5, 80, 0.25);
H.Numerator = b;
H.Denominator = a;
n = 0:100;
x = cos(0.2*pi*n)+sin(0.8*pi*n);
y = step(H,x);
% Filter states are nonzero
% Invoke step method again without resetting states
y1 = step(H,x);
isequal(y,y1) % returns 0
% Now reset filter states to 0
reset(H)
% Invoke step method
y2 = step(H,x);
isequal(y,y2) % returns a 1
```

Purpose   Filter input with IIR filter object

Syntax   Y = step(H,X)

Description Y = step(H,X) filters the real or complex input signal X using the IIR
               filter, H, to produce the output Y . When the input data is of a fixed-point
               type, it must be signed. The IIR filter object operates on each channel of
               the input signal independently over successive calls to step method.

               Note The object performs an initialization the first time the step
               method is executed. This initialization locks nontunable properties and
               input specifications, such as dimensions, complexity, and data type
               of the input data. If you change a nontunable property or an input
               specification, the System object issues an error. To change nontunable
               properties or inputs, you must first call the release method to unlock
               the object.

Purpose   Linear or FIR interpolation

Description   The Interpolator object interpolates real-valued inputs using linear
                   or polyphase FIR interpolation.

                   To interpolate real-valued inputs:

                   1 Define and set up your interpolation object. See "Construction" on
                        page 5-577.

                   2 Call step to interpolate the input according to the properties of
                        dsp.Interpolator. The behavior of step is specific to each object
                        in the toolbox.

Construction   H = dsp.Interpolator returns a handle to an interpolation object,
                   H. The interpolation method defaults to Linear and the interpolation
                   points default to a 5×1 vector.

                   H = dsp.Interpolator('*PropertyName*',*PropertyValue*,...)
                   returns an interpolation object, H, with the specified property name
                   and value pairs.

Properties   **InterpolationPointsSource**

                        Source of interpolation points

                        You can select a value of Property or Input port. If you set
                        this property to Input port, you must provide the interpolation
                        points as an input to the step method when you interpolate the
                        input. The default is Property.

                   InterpolationPoints

                        Interpolation points

                        This property only applies when you set the
                        InterpolationPointsSource property to Property.
                        Specify the interpolation points as a column vector or a matrix
                        with the same number of channels as the input.

dsp.Interpolator

The valid range of the values in the interpolation vector is from 1 to the number of samples in each channel of the input. If you specify interpolation points outside the valid range, the object *clips* the point to the nearest point in the valid range. For example, if the input is [2 3.1 -2.1], the valid range of interpolation points is from 1 to 3. If you specify the following vector of interpolation points [-1 1.5 2 2.5 3 3.5], the interpolator object clips -1 to 1 and 3.5 to 3. This clipping results in the interpolation points [1 1.5 2 2.5 3 3]. The default is [1.1;4.8;2.67;1.6;3.2]. This property is tunable.

Method

Interpolation method

Specify the interpolation method as Linear or FIR. When this property is Linear, the interpolator object interpolates data values by assuming that the data varies linearly between adjacent samples. When this property is FIR, the interpolator object uses polyphase interpolation to replace filtering (convolution) at the upsampled rate with a series of convolutions at the lower rate. The interpolator object always performs linear interpolation if there are insufficient low-rate samples to perform FIR interpolation as explained in the description of the FilterHalfLength property. The default is Linear.

FilterHalfLength

Interpolation filter half length

This property applies only when you set the Method property to FIR. For a filter half length of P, the polyphase FIR subfilters have length $2P$. FIR interpolation always requires $2P$ low-rate samples for every interpolation point.

- If the interpolation point does not correspond to a low-rate sample, FIR interpolation requires P low-rate samples below and P low-rate samples above the interpolation point.

- If the interpolation point corresponds to a low-rate sample, the 2*P*-sample requirement includes the low-rate sample.

- If there are less than *2P* neighboring low-rate samples, the interpolator object uses linear interpolation.

For example, for an input [1 4 1 4 1 4 1 4], upsampling by a factor of four results in the equally spaced interpolation points InterP = 1:0.25:8;. The points InterP(9:12) are [3.0 3.25 3.5 3.75]. Assuming a FilterHalfLength property equal to 2, interpolating at these points uses the four low-rate samples from the input with indices (2,3,4,5). If you set the FilterHalfLength property to 4 in this case, the interpolator object uses linear interpolation because there are not enough low-rate samples to perform FIR interpolation.

The longer the FilterHalfLength property, the better the quality of the interpolation. However, the costs are increased computation and requiring more low-rate samples below and above the interpolation point. In general, setting the FilterHalfLength property between 4 and 6 provides reasonably accurate interpolation.

InterpolationPointsPerSample

Interpolation points per input sample

This property only applies when you set the Method property to FIR and indicates the upsampling factor, *L*. An upsampling factor of *L* inserts *L*–1 zeros between low-rate samples. Interpolation results from filtering the upsampled sequence with a lowpass anti-imaging filter. The interpolator object uses a polyphase FIR implementation with InterpolationPointsPerSample subfilters of length 2*P*, where *P* is the FilterHalfLength property. Denoting the low-rate samples in the upsampled input by *nL*, *n=1,2,...*, the interpolator object uses exactly one of the InterpolationPointsPerSample subfilters, or filter arms, to interpolate at the points *nL+i/L*, where *i=0,1,2,...,L-1*.

If you specify interpolation points which do not correspond to a polyphase subfilter, the object rounds the point down to the nearest interpolation point associated with a polyphase subfilter. For example, you set the `InterpolationPointsPerSample` property to `4` and interpolate at the points `[3 3.2 3.4 3.6 3.8]`. The interpolator object uses the first polyphase subfilter for the points `[3.0 3.2]`, the second subfilter for the point `3.4`, the third subfilter for the point `3.6`, and the fourth subfilter for the point `3.8`.

Bandwidth

Normalized input bandwidth

This property applies only when you set the `Method` property to `FIR`. The normalized bandwidth is a real-valued scalar between 0 and 1, where 1 equals the Nyquist frequency, or 1/2 the sampling frequency (*Fs*). You may know that your input does not have frequency content above some cutoff frequency less than the Nyquist frequency. You can use this information to improve the FIR interpolation filters by relaxing the stopband requirements in frequency regions where the signal has no energy. For example, if the input signal does not have frequency content above *Fs/8*, you can specify a value of `0.25` for the `Bandwidth` property. The default value is `0.5`.

Methods		
	clone	Create interpolator object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Linear or FIR interpolation

Definitions   A polyphase implementation of an FIR interpolation filter *splits* the lowpass FIR filter impulse response into a number of different subfilters. Let *L* represent the number of interpolation points per sample, or the upsampling factor, and *P* the half length of the polyphase subfilters. Indexing from zero, if *h(n)* is the impulse response of the FIR filter, the *k*-th subfilter is:

$$h_k(n) = h(k+nL) \quad k = 0,1,\ldots,L-1 \quad n = 0,1,\ldots,2P-1$$

The following table describes the decomposition of an 18-coefficient FIR filter into 3 polyphase subfilters of length 6, the defaults for the FIR interpolator object:

Coefficients	**Polyphase Subfilter**
h(0),h(3),h(6), ..., h(15)	$h_0(n)$
h(1),h(4),h(7), ..., h(16)	$h_1(n)$
h(2),h(5),h(8), ..., h(17)	$h_2(n)$

The following code shows how to find the polyphase subfilters for the default FIR interpolator object:

```
H = dsp.Interpolator('Method','FIR');
L = H.InterpolationPointsPerSample;
P = H.FilterHalfLength;
FiltCoeffs = intfilt(L,P,H.Bandwith);
% Returns filter of length 2*P*L-1
FiltLen=length(FiltCoeffs);
FiltCols = ceil(FiltLen/2/L);
```

```
% We need 2*P*L coefficients
% Prepending a zero does not affect the filter magnitude
FiltCoeffs = [zeros(FiltCols*2*L-FiltLen,1); FiltCoeffs(:)];

% Each column of PolyPhaseCoeffs is a polyphase subfilter
PolyPhaseCoeffs  = reshape(FiltCoeffs,FiltCols,2*L)';
```

Examples   Compare linear interpolation with FIR interpolation:

```
x =[1 4];
x = repmat(x,1,4);
x1 = 1:0.25:8;
hFIR =dsp.Interpolator('Method','FIR','FilterHalfLength',2,...
'InterpolationPoints',x1','InterpolationPointsPerSample',4);
hLin =dsp.Interpolator('InterpolationPoints',x1');
OutFIR = step(hFIR,x');
OutLin = step(hLin,x');
stem(OutFIR,'b-.','linewidth',2); hold on;
stem(OutLin,'r','markerfacecolor',[1 0 0]);
axis([0 30 0 5]); legend('FIR','Linear','Location','Northeast');
```

For the indices 1 to 5 and 25 to 29, the interpolator object uses linear interpolation in both cases. The reason for this is that there are not enough low-rate samples surrounding the interpolation points at those indices to use FIR interpolation with the specified filter length.

Interpolate a sinusoid with linear interpolation:

```
t = 0:.0001:.0511;
x = sin(2*pi*20*t);
x1 = x(1:50:end);
I = 1:0.1:length(x1);
H = dsp.Interpolator('InterpolationPointsSource',...
'Input port');
y = H.step(x1',I');
stem(I',y, 'r');
title('Original and Interpolated Signal');
hold on; stem(x1, 'Linewidth', 2);
legend('Interpolated','Original');
```

dsp.Interpolator

Interpolate a sum of sinusoids with FIR interpolation and `Input Port` as the source of interpolation points:

```
Fs = 1000;
t = 0:(1/Fs):0.1-(1/Fs);
x = cos(2*pi*50*t)+0.5*sin(2*pi*100*t);
x1 = x(1:4:end);
I = 1:(1/4):length(x1);
% Decimate without aliasing
H = dsp.Interpolator('Method','FIR',...
'FilterHalfLength',3,'InterpolationPointsSource','Input Port');
y = H.step(x1',I);
stem(I,y,'r'); hold on;
axis([0 25 -2 2]);
stem(x1,'b','linewidth',2);
legend('Interpolated Signal','Original',...
'Location','Northeast');
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Interpolation block reference page. The object properties correspond to the Simulink block parameters, except:

Out of range interpolation points — The interpolator object only has the `Clip` option. The Simulink block has the additional `Clip and warn` and `Error` options.

See Also   intfilt | dsp.FIRInterpolator | dsp.VariableFractionalDelay

dsp.Interpolator.clone

Purpose   Create interpolator object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an interpolator object, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                  the step method.

                  The getNumInputs method returns a positive integer that is the number
                  of expected inputs (not counting the object itself) to the step method.
                  This value will change if you alter any properties that turn inputs on or
                  off. You must call the step method with the number of input arguments
                  equal to the result of getNumInputs(*H*).

dsp.Interpolator.getNumOutputs

Purpose　　　Number of outputs of step method

Syntax　　　`N = getNumOutputs(H)`

Description　`N = getNumOutputs(H)` returns the number of outputs, N, of `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the interpolator.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.Interpolator.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Linear or FIR interpolation

Syntax   Y = step(H,X)
                  Y = step(H,X,IPTS)

Description   Y = step(H,X) outputs the interpolated sequence, Y, of the input vector
                  or matrix X as specified in the InterpolationPoints property.

                  Y = step(H,X,IPTS) outputs the interpolated sequence as specified
                  by the input argument IPTS when the InterpolationPointsSource
                  property is set to 'Input port'. IPTS a column vector or a matrix of
                  interpolation points. If IPTS is a matrix, it must have the same number
                  of channels as the input.

                  Note The object performs an initialization the first time the step
                  method is executed. This initialization locks nontunable properties and
                  input specifications, such as dimensions, complexity, and data type
                  of the input data. If you change a nontunable property or an input
                  specification, the System object issues an error. To change nontunable
                  properties or inputs, you must first call the release method to unlock
                  the object.

dsp.LDLFactor

Purpose   Factor square Hermitian positive definite matrices into components

Description   The `LDLFactor` object factors square Hermitian positive definite
matrices into lower, upper, and diagonal components. The object uses
only the lower triangle of S.

To factor these matrices into lower, upper, and diagonal components:

1 Define and set up your LDL factor object. See "Construction" on
page 5-592.

2 Call `step` to factor the matrices according to the properties of
`dsp.LDLFactor`. The behavior of `step` is specific to each object in
the toolbox.

Construction   H = `dsp.LDLFactor` returns an LDL factor System object, H, that
computes unit lower triangular L and diagonal D such that S = LDL' for
square, symmetric/Hermitian, positive definite input matrix S.

H = `dsp.LDLFactor('`*PropertyName*`',`*PropertyValue*`,...)` returns
an LDL factor System object, H, with each specified property set to the
specified value.

Properties   **Fixed-Point Properties**

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Full precision`,
`Same as input`, `Same as product` or `Custom`. The default is `Full
precision`

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype`
object with a `Signedness` of `Auto`. This property applies when you

set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

CustomIntermediateProductDataType

Intermediate product word and fraction lengths

Specify the intermediate product fixed-point type as a signed, scaled `numerictype` object. This property applies when you set the `IntermediateProductDataType` property to `Custom`. The default is `numerictype(true,16,15)`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numerictype([],32,30)`.

IntermediateProductDataType

Intermediate product word and fraction lengths

Specify the intermediate product fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of Wrap or Saturate. The default is Wrap.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as Full precision, Same as input or Custom. The default is Full precision.

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as: Ceiling, Convergent, Floor, Nearest, Round, Simplest or Zero. The default is Floor.

Methods	clone	Create an LDL Factor object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs for step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	step	Decompose matrix into components

Examples   Decompose a square Hermitian positive definite matrix using LDL factor:

```
A = gallery('randcorr',5);
hldl = dsp.LDLFactor;
```

```
y = step(hldl, A);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the LDL Factorization block reference page. The object properties correspond to the block parameters, except:

No object property that corresponds to the **Non-positive definite input** block parameter. The object does not issue any alerts for nonpositive definite inputs. The output is not a valid factorization. A partial factorization is in the upper left corner of the output.

See Also   dsp.LUFactor

dsp.LDLFactor.clone

Purpose   Create an LDL Factor object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates an LDLFactor System object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   `N = getNumInputs(H)`

Description   `N = getNumInputs(H)` returns the number of expected inputs, `N`, to the `step` method

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(`*H*`)`.

dsp.LDLFactor.getNumOutputs

Purpose   Number of outputs for step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                     method

                     The getNumOutputs method returns a positive integer that is the
                     number of outputs from the step method. This value will change if you
                     alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the LDLFactor object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.LDLFactor.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Decompose matrix into components

Syntax   Y = step(H,S)

Description Y = step(H,S) decomposes the matrix S into lower, upper, and diagonal components. The output Y is a composite matrix with the L as its lower triangular part and D as the diagonal and L' as its upper triangular part. If S is not positive definite the output Y is not a valid factorization.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

---

dsp.LevinsonSolver

Purpose   Solve linear system of equations using Levinson-Durbin recursion

Description The `LevinsonSolver` object solves linear systems of equations using Levinson-Durbin recursion.

To solve linear systems of equations using Levinson-Durbin recursion:

1 Define and set up your System object. See "Construction" on page 5-602.

2 Call `step` to solve the system of equations according to the properties of `dsp.LevinsonSolver`. The behavior of `step` is specific to each object in the toolbox.

Construction `H = dsp.LevinsonSolver` returns a System object, H, that solves a Hermitian Toeplitz system of equations using the Levinson-Durbin recursion.

`H = dsp.LevinsonSolver('PropertyName',PropertyValue,...)` returns a Levinson-Durbin object, H, with each specified property set to the specified value.

Properties   **AOutputPort**

Enable polynomial coefficients output

Set this property to `true` to output the polynomial coefficients *A*. Both `AOutputPort` and `KOutputPort` properties cannot be `false` at the same time. For scalar inputs, set the `AOutputPort` property to `true`. The default is `false`.

KOutputPort

Enable reflection coefficients output

Set this property to `true` to output the reflection coefficients *K*. You cannot set both the `AOutputPort` and `KOutputPort` properties to `false` at the same time. For scalar inputs, you must set the `KOutputPort` property to `false`. The default is `true`.

PredictionErrorOutputPort

Enable prediction error output

Set this property to `true` to output the prediction error. The default is `false`.

ZerothLagZeroAction

Action when value of lag zero is zero

Specify the output for an input with the first coefficient as zero. Select `Ignore` or `Use zeros`. The default is `Use zeros`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap`, `Saturate`. The default is `Wrap`.

ACoefficientDataType

A coefficient word and fraction lengths

This constant property has a value of `Custom`.

CustomACoefficientDataType

A coefficient word and fraction lengths

Specify the *A* coefficient fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. The default is `numerictype([],16,15)`.

KCoefficientDataType

K coefficient word and fraction lengths

This constant property has a value of `Custom`.

CustomKCoefficientDataType

K coefficient word and fraction lengths

Specify the *K* coefficient fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. The default is `numerictype([],16,15)`.

PredictionErrorDataType

Prediction error power word and fraction lengths

Specify the prediction error power fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

CustomPredictionErrorDataType

Prediction error power word and fraction lengths

Specify the prediction error power fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `PredictionErrorDataType` property is `Custom`. The default is `numerictype([],16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as `Same as input` or `Custom`. The default is `Custom`

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `ProductDataType` property is `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the Accumulator fixed-point data type as `Same as input`, `Same as product`, or `Custom`. The default is `Custom`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when the `AccumulatorDataType` property is `Custom`. The default is `numerictype([],32,30)`.

Methods

clone	Create Levinson solver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Reflection coefficients corresponding to columns of input

Examples

Use the Levinson solver to compute polynomial coefficients from autocorrelation coefficients:

```
hlevinson = dsp.LevinsonSolver;
hlevinson.AOutputPort = true;
hlevinson.KOutputPort = false;
x = (1:100)';
hac = dsp.Autocorrelator(...
'MaximumLagSource', 'Property', ...
'MaximumLag', 10);
```

```
 a = step(hac, x);
 c = step(hlevinson, a); % Compute polynomial coefficients
```

Algorithms   This object implements the algorithm, inputs, and outputs described
                   on the Levinson-Durbin block reference page. The object properties
                   correspond to the block parameters, except:

    Output(s) block parameter corresponds to the AOutputPort and the
    KOutputPort object properties.

See Also   dsp.Autocorrelator

Purpose   Create Levinson solver object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a LevinsonSolver object System object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step
method

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

dsp.LevinsonSolver.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the LevinsonSolver System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.LevinsonSolver.step

Purpose   Reflection coefficients corresponding to columns of input

Syntax   K = step(H,X)
                 A = step(H,X)
                 [A, K] = step(H,X)
                 [..., P] = step(H,X)

Description K = step(H,X) returns reflection coefficients K corresponding to the
                 columns of input X. X is typically a column or matrix of autocorrelation
                 coefficients with lag 0 as the first element.

                 A = step(H,X) returns polynomial coefficients A when the AOutputPort
                 property is true and the KOutputPort property is false.

                 [A, K] = step(H,X) returns polynomial coefficients A and reflection
                 coefficients K when both the AOutputPort and KOutputPort properties
                 are true.

                 [..., P] = step(H,X) also returns the error power P when the
                 PredictionErrorOutputPort property is true.

                 Note The object performs an initialization the first time the step
                 method is executed. This initialization locks nontunable properties and
                 input specifications, such as dimensions, complexity, and data type
                 of the input data. If you change a nontunable property or an input
                 specification, the System object issues an error. To change nontunable
                 properties or inputs, you must first call the release method to unlock
                 the object.

Purpose   LMS adaptive filter

Description   The LMSFilter implements an adaptive FIR filter object that returns
the filtered output, the error vector, and filter weights. The LMS filter
uses one of five different LMS algorithms.

To implement the adaptive FIR filter object:

1 Define and set up your adaptive FIR filter object. See "Construction"
on page 5-613.

2 Call step to implement the filter according to the properties of
dsp.LMSFilter. The behavior of step is specific to each object in
the toolbox.

Construction   H = dsp.LMSFilter returns an adaptive FIR filter object, H, that
computes the filtered output, filter error and the filter weights for a
given input and desired signal using the Least Mean Squares (LMS)
algorithm.

H = dsp.LMSFilter('*PropertyName*', *PropertyValue*,...) returns
an LMS filter object, H, with each property set to the specified value.

H = dsp.LMSFilter(LEN,'*PropertyName*',*PropertyValue*,...)
returns an LMS filter object, H, with the Length property set to LEN, and
other specified properties set to the specified values.

Properties   **Method**

Method to calculate filter weights

Specify the method used to calculate filter weights as LMS,
Normalized LMS, Sign-Error LMS, Sign-Data LMS, or Sign-Sign
LMS. The default is LMS.

Length

Length of FIR filter weights vector

Specify the length of the FIR filter weights vector as a positive integer. The default is 32.

StepSizeSource

How to specify adaptation step size

Choose how to specify the adaptation step size factor as `Property` or `Input port`. The default is `Property`.

StepSize

Adaptation step size

Specify the adaptation step size factor as a nonnegative real number. For convergence of the normalized LMS method, set the step size greater than `0` and less than `2`. This property only applies when the `StepSizeSource` property is `Property`. The default is `0.1`. This property is tunable.

LeakageFactor

Leakage factor used in LMS filter

Specify the leakage factor as a real number between `0` and `1` inclusive. A leakage factor of `1` corresponds to no leakage in the adapting method. The default is `1`. This property is tunable.

InitialConditions

Initial conditions of filter weights

Specify the initial values of the FIR filter weights as a scalar or vector of length equal to the `Length` property value. The default is `0`.

AdaptInputPort

Enable weight adaptation

Specify when the LMS filter should adapt the filter weights. By default, the value of this property is `false`, and the object continuously updates the filter weights. When this property is set to `true`, an adaptation control input is provided to the

step method. If the value of this input is nonzero, the object continuously updates the filter weights. If the input is zero, the filter weights remain at their current value.

WeightsResetInputPort

Enable weight reset

Specify when the LMS filter should reset the filter weights. By default, the value of this property is false, and the object does not reset the weights. When this property is set to true, a reset control input is provided to the step method, and the WeightsResetCondition property applies. The object resets the filter weights based on the values of the WeightsResetCondition property and the reset input to the step method.

WeightsResetCondition

Reset trigger setting for filter weights

Specify the event to reset the filter weights as Rising edge, Falling edge, Either edge, or Non-zero. The LMS filter resets the filter weights based on the values of this property and the reset input to the step method. This property only applies when the WeightsResetInputPort property is true. The default is Non-zero.

WeightsOutputPort

Enable returning filter weights

Set this property to true to output the adapted filter weights. The default is true.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as Ceiling, Convergent, Floor, Nearest, Round, Simplest, or Zero. The default is Floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

StepSizeDataType

Step size word and fraction lengths

Specify the step size fixed-point data type as `Same word length as first input` or `Custom`. Setting this property also sets the `LeakageFactorDataType` property to the same value. This property only applies when the `StepSizeSource` property is `Property`. The default is `Same word length as first input`.

CustomStepSizeDataType

Step size word and fraction lengths

Specify the step size fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `StepSizeSource` property is `Property` and the `StepSizeDataType` property is `Custom`. Thhis property and the `CustomLeakageFactorDataType` property must have the same word lengths. The default is `numerictype([],16,15)`.

LeakageFactorDataType

Leakage factor word and fraction lengths

Specify the leakage factor fixed-point data type as `Same word length as first input` or `Custom`. Setting this property also sets the `StepSizeDataType` property to the same value. This property only applies when the `StepSizeSource` property is `Property`. The default is `Same word length as first input`.

CustomLeakageFactorDataType

Leakage factor word and fraction lengths

Specify the leakage factor fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property only applies

when the `StepSizeSource` property is `Property` and the `LeakageFactorDataType` property is `Custom`. This property and the `CustomStepSizeDataType` property must have the same word lengths. The default is `numerictype([],16,15)`.

WeightsDataType

Weights word and fraction lengths

Specify the filter weights fixed-point data type as `Same as first input` or `Custom`. The default is `Same as first input`.

CustomWeightsDataType

Weights word and fraction lengths

Specify the filter weights fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `WeightsDataType` property is `Custom`. The default is `numerictype([],16,15)`.

EnergyProductDataType

Energy product word and fraction lengths

Specify the energy product fixed-point data type as `Same as first input` or `Custom`. This property only applies when the `Method` property is `Normalized LMS`. Setting this property also sets the `ConvolutionProductDataType`, `StepSizeErrorProductDataType`, `WeightsUpdateProductDataType`, and `QuotientDataType` properties to the same value. The default is `Same as first input`.

CustomEnergyProductDataType

Energy product word and fraction lengths

Specify the energy product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `Method` property is `Normalized LMS` and the `EnergyProductDataType` property is `Custom`. The `CustomEnergyProductDataType`,

CustomConvolutionProductDataType,
CustomStepSizeErrorProductDataType,
CustomWeightsUpdateProductDataType, and
CustomQuotientDataType properties must have the same word
lengths. The default is numerictype([],32,20).

EnergyAccumulatorDataType

Energy accumulator word and fraction lengths

Specify the energy accumulator fixed-point data type as Same as
first input or Custom. This property only applies when the
Method property is Normalized LMS. Setting this property also
sets the ConvolutionAccumulatorDataType property to the same
value. The default is Same as first input.

CustomEnergyAccumulatorDataType

Energy accumulator word and fraction lengths

Specify the energy accumulator fixed-point type as a
scaled numerictype object with a Signedness of Auto.
This property only applies when the Method property is
Normalized LMS and the EnergyAccumulatorDataType
property is Custom. The CustomEnergyProductDataType,
CustomConvolutionProductDataType,
CustomStepSizeErrorProductDataType,
CustomWeightsUpdateProductDataType, and
CustomQuotientDataType properties must have the same word
lengths. The default is numerictype([],32,20).

ConvolutionProductDataType

Convolution product word and fraction lengths

Specify the convolution product fixed-point data type as Same
as first input or Custom. Setting this property also sets the
EnergyProductDataType, StepSizeErrorProductDataType,
WeightsUpdateProductDataType and QuotientDataType
properties to the same value. The default is Same as first
input.

CustomConvolutionProductDataType

Convolution product word and fraction lengths

Specify the convolution product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property only applies when the ConvolutionProductDataType property is Custom. The CustomEnergyProductDataType, CustomConvolutionProductDataType, CustomStepSizeErrorProductDataType, CustomWeightsUpdateProductDataType, and CustomQuotientDataType properties must have the same word lengths. The default is numerictype([],32,20).

ConvolutionAccumulatorDataType

Convolution accumulator word and fraction lengths

Specify the convolution accumulator fixed-point data type as Same as first input or Custom. Setting this property also sets the EnergyAccumulatorDataType property to the same value. The default is Same as first input.

CustomConvolutionAccumulatorDataType

Convolution accumulator word and fraction lengths

Specify the convolution accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property only applies when the ConvolutionAccumulatorDataType property is Custom. The CustomEnergyAccumulatorDataType and CustomConvolutionAccumulatorDataType properties must have the same word lengths. The default isnumerictype([],32,20).

StepSizeErrorProductDataType

Step size error product word and fraction lengths

Specify the step size error product fixed-point data type as Same as first input or Custom. Setting this property also sets the ConvolutionProductDataType, EnergyProductDataType, WeightsUpdateProductDataType, and QuotientDataType

properties to the same value. The default is `Same as first input`.

CustomStepSizeErrorProductDataType

Step size error product word and fraction lengths

Specify the step size error product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `StepSizeErrorProductDataType` property is `Custom`. The `CustomEnergyProductDataType`, `CustomConvolutionProductDataType`, `CustomStepSizeErrorProductDataType`, `CustomWeightsUpdateProductDataType`, and `CustomQuotientDataType` properties must have the same word lengths. The default is `numerictype([],32,20)`.

WeightsUpdateProductDataType

Weight update product word and fraction lengths

Specify the weight update product fixed-point data type as `Same as first input` or`Custom`. Setting this property also sets the `ConvolutionProductDataType`, `EnergyProductDataType`, `StepSizeErrorProductDataType`, and `QuotientDataType` properties to the same value. The default is `Same as first input`.

CustomWeightsUpdateProductDataType

Weight update product word and fraction lengths

Specify the weight update product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `WeightsUpdateProductDataType` property is `Custom`. The `CustomEnergyProductDataType`, `CustomConvolutionProductDataType`, `CustomStepSizeErrorProductDataType`, `CustomWeightsUpdateProductDataType`, and `CustomQuotientDataType` properties must have the same word lengths. The default is `numerictype([],32,20)`.

QuotientDataType

Quotient word and fraction lengths

Specify the quotient fixed-point data type as `Same as first input`, `Custom`. This property only applies when the `Method` property is `Normalized LMS`. Setting this property also sets the `ConvolutionProductDataType`, `EnergyProductDataType`, `StepSizeErrorProductDataType`, and `WeightsUpdateProductDataType` properties to the same value. The default is `Same as first input`.

CustomQuotientDataType

Quotient word and fraction lengths

Specify the quotient fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `Method` property is `Normalized LMS` and the `QuotientDataType` property is `Custom`. The `CustomEnergyProductDataType`, `CustomConvolutionProductDataType`, `CustomStepSizeErrorProductDataType`, `CustomWeightsUpdateProductDataType`, and `CustomQuotientDataType` properties must have the same word lengths. The default is `numerictype([],32,20)`.

Methods		
	clone	Create LMS filter object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes

dsp.LMSFilter

Examples   Determine system identification of an FIR filter:

```
hlms1 = dsp.LMSFilter(11, 'StepSize', 0.01);
hfilt = dsp.DigitalFilter; % System to be identified
hfilt.TransferFunction = 'FIR (all zeros)';
hfilt.Numerator = fir1(10, .25);
x = randn(1000,1); % input signal

d = step(hfilt, x) + 0.01*randn(1000,1); % desired signal
[y,e,w] = step(hlms1, x, d);

subplot(2,1,1), plot(1:1000, [d,y,e]);
title('System Identification of an FIR filter');
legend('Desired', 'Output', 'Error');
xlabel('time index'); ylabel('signal value');
subplot(2,1,2); stem([hfilt.Numerator.', w]);
legend('Actual','Estimated');
xlabel('coefficient #'); ylabel('coefficient value');
```

Cancel noise in a signal:

```
hlms2 = dsp.LMSFilter('Length', 11, ...
   'Method', 'Normalized LMS',...
   'AdaptInputPort', true, ...
   'StepSizeSource', 'Input port', ...
   'WeightsOutputPort', false);
hfilt2 = dsp.DigitalFilter(...
   'TransferFunction', 'FIR (all zeros)', ...
   'Numerator', fir1(10, [.5, .75]));
x = randn(1000,1); % Noise
d = step(hfilt2, x) + sin(0:.05:49.95)'; % Noise + Signal
```

```
a = 1; % adaptation control
mu = 0.05; % step size
[y, err] = step(hlms2, x, d, mu, a);

subplot(2,1,1), plot(d), title('Noise + Signal');
subplot(2,1,2),plot(err), title('Signal');
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the LMS Filter block reference page. The object properties correspond to the block parameters.

See Also   dsp.BlockLMSFilter | dsp.DigitalFilter

dsp.LMSFilter.clone

Purpose   Create LMS filter object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an LMS filter object, C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.LMSFilter.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the LMS filter.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.LMSFilter.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset filter states for LMS filter

Syntax   reset(H)

Description   reset(H) resets the filter states of the LMS FIR filter,H, to their initial values specified in the InitialConditions property. The initial filter state values correspond to the initial conditions for difference equation defining the adaptive filter. After the step method applies the LMS filter to nonzero input data, the states may be different. Invoking the step method again without first invoking thereset method may produce different outputs for an identical input.

Example of resetting filter states:

```
hlms1 = dsp.LMSFilter(11, 'StepSize', 0.01);
hfilt = dsp.DigitalFilter; % System to be identified
hfilt.TransferFunction = 'FIR (all zeros)';
hfilt.Numerator = fir1(10, .25);
x = randn(1000,1); % input signal
d = step(hfilt, x) + 0.01*randn(1000,1); % desired signal
[y,e,w] = step(hlms1, x, d);
% Call step method again without resetting filter states
[y1,e1,w1] = step(hlms1,x,d);
isequal(y,y1) % Returns 0
% Now reset the filter states to zero
reset(hlms1)
% invoke step method
[y2,e2,w2] = step(hlms1,x,d);
isequal(y,y2) % Returns 1
```

dsp.LMSFilter.step

Purpose Apply LMS adaptive filter to input

Syntax
```
[Y,ERR,WTS] = step(H,X,D)
[Y, ERR] = step(H,X,D)
[...] = step(H,X,D,MU)
[...] = step(H,X,D,A)
[...] = step(H,X,D,R)
[Y,ERR,WTS] = step(H,X,D,MU,A,R)
```

Description `[Y,ERR,WTS] = step(H,X,D)` applies the LMS filter object, `H` to the input `X`, using `D` as the desired signal. This approach returns the filtered output in `Y`, the filter error in `ERR`, and the estimated filter weights in `WTS`. The LMS filter estimates the filter weights needed to minimize the mean square error between the output and the desired signal.

`[Y, ERR] = step(H,X,D)` filters the input `X`, using `D` as the desired signal. This approach returns the filtered output in `Y` and the filter error in `ERR` when the `WeightsOutputPort` property is `false`.

`[...]  = step(H,X,D,MU)` uses `MU` as the step size, when the `StepSizeSource` property is `'Input port'`.

`[...]  = step(H,X,D,A)` uses `A` as the adaptation control, when the `AdaptInputPort` property is `true`. When `A` is nonzero, the LMS filter continuously updates the filter weights. When `A` is zero, the filter weights remain constant.

`[...]  = step(H,X,D,R)` uses `R` as a reset signal when the `WeightsResetInputPort` property is `true`. The `WeightsResetCondition` property can be used to set the reset trigger condition. If a reset event occurs, the LMS filter resets the filter weights to their initial values.

`[Y,ERR,WTS] = step(H,X,D,MU,A,R)` filters the input `X` using `D` as the desired signal, `MU` as the step size, `A` as the adaptation control, and `R` as the reset signal. This approach returns the filtered output in `Y`, the filter error in `ERR`, and the adapted filter weights in `WTS`.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

dsp.LowerTriangularSolver

Purpose   Solve lower-triangular matrix equation

Description   The LowerTriangularSolver object solves $LX = B$ for X when L is a square, lower-triangular matrix with the same number of rows as B.

To solve $LX = B$ for X:

1 Define and set up your linear system solver. See "Construction" on page 5-632.

2 Call step to solve the equation according to the properties of dsp.LowerTriangularSolver. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.LowerTriangularSolver returns a linear system solver, H, used to solve the linear system $LX = B$, where L is a lower (or unit-lower) triangular matrix.

H = dsp.LowerTriangularSolver('*PropertyName*',*PropertyValue*,...) returns a linear system solver, H, with each specified property set to the specified value.

Properties   **OverwriteDiagonal**

Replace diagonal elements of input with ones

When you set this property to true, the linear system solver replaces the elements on the diagonal of the input, *L*, with ones. Set this property to either true or false. The default is false.

RealDiagonalElements

Indicate that diagonal of complex input is real

When you set this property to true, the linear system solver optimizes computation speed if the diagonal elements of complex input, *L*, are real. This property applies only when you set the OverwriteDiagonal property to false. Set this property to either true or false. The default is false.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

ProductDataType

Data type of product

Specify the product data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `ProductDataType` property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as `Full precision`, `Same as first input`, `Same as product`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Data type of output

Specify the output data type as `Same as first input` or `Custom`. The default is `Same as first input`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

Methods

clone	Create lower triangular solver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Solve matrix equation for specified inputs

Examples

Solve a lower-triangular matrix equation:

```
hlowtriang = dsp.LowerTriangularSolver;
```

```
u = tril(rand(4, 4));
b = rand(4, 1);
% Check that result is the solution to the linear
% equations.
x1 = inv(u)*b
x = step(hlowtriang, u, b)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on
                   the Forward Substitution block reference page. The object properties
                   correspond to the block parameters.

See Also   dsp.UpperTriangularSolver

dsp.LowerTriangularSolver.clone

Purpose   Create lower triangular solver object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a `LowerTriangularSolver` object C, with the same property values as H. The `clone` method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.LowerTriangularSolver.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the LowerTriangularSolver
                    object H.

                    The isLocked method returns a logical value that indicates whether
                    input attributes and nontunable properties for the object are locked. The
                    object performs an internal initialization the first time the step method
                    is executed. This initialization locks nontunable properties and input
                    specifications, such as dimensions, complexity, and data type of the
                    input data. After locking, the isLocked method returns a true value.

dsp.LowerTriangularSolver.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Solve matrix equation for specified inputs

Syntax   X = step(H,L,B)

Description   X = step(H,L,B) computes the solution, *X*, of the matrix equation
                     $LX = B$, where *L* is a square, lower-triangular matrix with the same
                     number of rows as the matrix *B*.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.LPCToAutocorrelation

Purpose	Convert linear prediction coefficients to autocorrelation coefficients
Description	The `LPCToAutocorrelation` System object converts linear prediction coefficients to autocorrelation coefficients.

To convert LPC to autocorrelation coefficients:

1 Define and set up your LPC to autocorrelation object. See "Construction" on page 5-642.

2 Call `step` to convert LPC according to the properties of `dsp.LPCToAutocorrelation`. The behavior of `step` is specific to each object in the toolbox.

Construction	`H = dsp.LPCToAutocorrelation` returns an LPC to autocorrelation System object, `H`, that converts linear prediction coefficients (LPC) to autocorrelation coefficients.

`H = dsp.LPCToAutocorrelation('`*PropertyName*`',`*PropertyValue*`,...)` returns an LPC to autocorrelation conversion object, `H`, with each specified property set to the specified value.

Properties

PredictionErrorInputPort

Enable prediction error power input

Choose how to select the prediction error power. When you set this property to `true`, you must specify the prediction error power as a second input to the `step` method. When you set this property to `false`, the object assumes that the prediction error power is 1. The default is `false`.

NonUnityFirstCoefficientAction

Action to take when first LPC coefficient is not 1

Specify the action that the object takes when the first coefficient of each channel of the LPC input is not 1. Select `Replace with 1` or `Normalize`. The default is `Replace with 1`.

Methods	clone	Create LPC to autocorrelation object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	step	Autocorrelation coefficients from LPC coefficents

Examples
Convert the linear prediction coefficients to autocorrelation coefficients:

```
a = [1.0 -1.4978 1.4282 -1.3930 0.9076 -0.3855 0.0711].';
hlpc2ac = dsp.LPCToAutocorrelation;
ac = step(hlpc2ac, a);
```

Algorithms
This object implements the algorithm, inputs, and outputs described on the LPC/RC to Autocorrelation block reference page. The object properties correspond to the block parameters, except:

The object does not have a property that corresponds to the **Type of Conversion** block parameter. The object's behavior corresponds to the block's behavior when you set the **Type of Conversion** parameter to LPC to autocorrelation.

See Also
dsp.LPCToLSF | dsp.LPCToRC | dsp.LPCToCepstral | dsp.RCToAutocorrelation

dsp.LPCToAutocorrelation.clone

Purpose   Create LPC to autocorrelation object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an LPCToAutocorrelation System object C,
                   with the same property values as H. The clone method creates a new
                   unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.LPCToAutocorrelation.getNumOutputs

Purpose Number of outputs of step method

Syntax N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the LPCToAutocorrelation
System object H.

The isLocked method returns a logical value that indicates whether
input attributes and nontunable properties for the object are locked. The
object performs an internal initialization the first time the step method
is executed. This initialization locks nontunable properties and input
specifications, such as dimensions, complexity, and data type of the
input data. After locking, the isLocked method returns a true value.

dsp.LPCToAutocorrelation.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

> **Note** You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose　　Autocorrelation coefficients from LPC coefficents

Syntax　　`AC = step(H,A)`
　　　　　　　`AC = step(H,A,P)`

Description　　`AC = step(H,A)` converts the columns of the linear prediction coefficients, `A`, to autocorrelation coefficients, `AC`. The object assumes a prediction error power of 1.

　　　　　　`AC = step(H,A,P)` when you set the `PredictionErrorInputPort` property to `true`, converts the columns of the linear prediction coefficients, `A`, to autocorrelation coefficients, `AC`. This conversion uses `P` as the prediction error power. `P` must be a row vector with same number of columns as `A`.

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.LPCToCepstral

Purpose　　　Convert linear prediction coefficients to cepstral coefficients

Description　　The `LPCToCepstral` object converts linear prediction coefficients to cepstral coefficients.

To convert LPC to cepstral coefficients:

1 Define and set up your LPC to cepstral converter. See "Construction" on page 5-650.

2 Call `step` to convert LPC according to the properties of `dsp.LPCToCepstral`. The behavior of `step` is specific to each object in the toolbox.

Construction　　`H = dsp.LPCToCepstral` returns an LPC to cepstral converter object, `H`, that converts linear prediction coefficients (LPCs) to cepstral coefficients (CCs).

`H = dsp.LPCToCepstral('`*PropertyName*`',`*PropertyValue*`,...)` returns an LPC to cepstral converter object, `H`, with each specified property set to the specified value.

Properties　　**PredictionErrorInputPort**

Enable prediction error power input

Choose how to set the prediction error power. When you set this property to `true`, you must specify the prediction error as a second input to the `step` method. When you set this property to `false`, the oajbect assumes the prediction error power is 1. The default is `false`.

CepstrumLengthSource

Source of cepstrum length

Select how to specify the length of cepstral coefficients: `Auto` or `Property`. The default is `Auto`. When this property is set to `Auto`, the length of each channel of the cepstral coefficients output is the

same as the length of each channel of the input LPC coefficients. The default is `Property`.

CepstrumLength

Number of output cepstral coefficients

Set the length of the output cepstral coefficients vector as a scalar numeric integer. This property applies when you set the `CepstrumLengthSource` property to `Property`. The default is `10`.

NonUnityFirstCoefficientAction

LPC coeffecent nonunity action

Specify the action that the object takes when the first coefficient of each channel of the LPC input is not `1`. Select `Replace with 1` or `Normalize`. The default is `Replace with 1`.

Methods

clone	Create LPC to cepstral object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Cepstral coefficients from columns of input LPC coefficients

Examples

Convert the linear prediction coefficients (LPC) to cepstral coefficients:

```
hlevinson = dsp.LevinsonSolver;
hlevinson.AOutputPort = true; % Output polynomial coefficients
hac = dsp.Autocorrelator;
```

```
hac.MaximumLagSource = 'Property';
hac.MaximumLag = 9; % Compute autocorrelation lags between [0:9]
hlpc2cc = dsp.LPCToCepstral;
x = [1:100]';
a = step(hac, x);
A = step(hlevinson, a); % Compute LPC coefficients
CC = step(hlpc2cc, A); % Convert LPC to CC.
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the LPC to/from Cepstral Coefficients block reference page. The object properties correspond to the block parameters, except:

The object does not have a property that corresponds to the **Type of Conversion** block parameter. The object's behavior corresponds to the block's behavior when you set the **Type of Conversion** parameter to LPCs to cepstral coefficients.

See Also   dsp.CepstralToLPC | dsp.LPCToLSF | dsp.LPCToRC

Purpose　　　Create LPC to cepstral object with same property values

Syntax　　　　C = clone(H)

Description　　C = clone(H) creates an LPCToCepstral System object C, with the same property values as H. The clone method creates a new unlocked object.

dsp.LPCToCepstral.getNumInputs

Purpose	Number of expected inputs to step method
Syntax	N = getNumInputs(H)
Description	N = getNumInputs(H) returns the number of expected inputs, N, to the step method

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   `N = getNumOutputs(H)`

Description   `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

dsp.LPCToCepstral.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the LPCToCepstral System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose	Cepstral coefficients from columns of input LPC coefficients
Syntax	CC = step(H,A) CC = step(H,A,P)
Description	CC = step(H,A) computes the cepstral coefficients, CC, from the columns of input linear prediction coefficients, A. The object assumes the prediction error power is 1.

CC = step(H,A,P) when you set the PredictionErrorInputPort property to true, computes the cepstral coefficients, CC, from the columns of input linear prediction coefficients, A. This conversion uses P as the prediction error power.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Convert linear prediction coefficients to line spectral frequencies

Description   The LPCToLSF object converts linear prediction coefficients to line spectral frequencies.

To convert LPC to LSF:

1 Define and set up your LPC to LSF converter. See "Construction" on page 5-659.

2 Call step to convert LPC according to the properties of dsp.LPCToLSF. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.LPCToLSF returns a System object, H, that converts linear prediction coefficients (LPCs) to line spectral frequencies (LSFs).

H = dsp.LPCToLSF(*'PropertyName'*,*PropertyValue*,...) returns an LPC to LSF System object, H, with each specified property set to the specified value.

Properties   **NumCoarseGridPoints**

Number of coarse subintervals used for finding roots (LSP values)

Specify the number of coarse subintervals, n, used for finding line spectral pairs (LSP) values as a positive scalar integer. LSPs, which are the roots of two particular polynomials related to the input LPC polynomial, always lie in the range (–1, 1). The System object finds these roots using the Chebyshev polynomial root finding method. To compute LSF outputs, the object computes the arc cosine of the LSPs, outputting values ranging from 0 to pi radians. The object divides the interval (–1, 1) into n subintervals and looks for roots in each subinterval. If you set *n* too small in relation to the LPC polynomial order, the object can fail to find some of the roots. The default is 64. This property is tunable.

NumBisects

Value of bisection refinement used for finding roots

Specify the root bisection refinement value, k, used in the Chebyshev polynomial root finding method, where each line spectral pair (LSP) output is within

$$\frac{1}{(n \cdot 2^k)}$$

of the actual LSP value. Here n is the value of the NumCoarseGridPoints property, and the object searches a maximum of $k \cdot (n-1)$ points for finding the roots. You must set the NumBisects property value k, to a positive scalar integer. The default is 4. This property is tunable.

ExceptionOutputPort

Produces output with validity status of LSF output

Set this property to true to return a second output that indicates whether the computed LSF values are valid. The output is a vector with a length equal to the number of channels. A logical value of 1 indicates valid output. A logical value of 0 indicates invalid output. The LSF outputs are invalid when the object fails to find all the LSF values or when the input LPCs are unstable. The default is false.

OverwriteInvalidOutput

Enable overwriting invalid output with previous output

Specify the action that the System object should take for invalid LSF outputs. When you set this property to true, the object overwrites the invalid output with the previous output. When you set this property to false, the object does not take any action on invalid outputs and ignores the outputs.

FirstOutputValuesSource

Source of values for first output when output is invalid

Specify the source of values for the first output when the output is invalid as Auto or Property. This property applies when you

set the OverwriteInvalidOutput property to true. The default is Auto. When you set this property to Auto, the object uses a default value for the first output. The default value corresponds to the LSF representation of an allpass filter.

FirstOutputValues

Value of the first output

Specify a numeric vector of LSF values for overwriting an invalid first output. The length of this vector must be one less than the length of the input LPC vector. For multichannel inputs, you can set this property to a matrix with the same number of channels as the input, or one vector that is applied to every channel. The default is an empty vector. This property applies when you set the OverwriteInvalidOutput property to true and the FirstOutputValuesSource property to Property.

NonUnityFirstCoefficientAction

Action to take when first LPC coefficient is not 1

Specify the action the object takes when the first coefficient of each channel of the LPC input is not 1 as Replace with 1 or Normalize. The default is Replace with 1.

Methods

clone	Create LPC To LSF object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset		Reset values for overwriting invalid outputs to their initial values
step		Convert LPC coefficients to line spectral frequencies

Examples   Convert to linear prediction coefficients to line spectral frequencies:

```
a = [1.0000 0.6149 0.9899 0.0000 0.0031 -0.0082]';
hlpc2lsf = dsp.LPCToLSF;
y = step(hlpc2lsf, a); % Convert to LSF coefficients
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the LPC to LSF/LSP Conversion block reference page. The object properties correspond to the block parameters, except:

There is no object property that corresponds to the **Output** block parameter. The object only supports LSF outputs in the range $(0, \prod)$

See Also   dsp.LSFToLPC | dsp.LPCToLSP

Purpose   Create LPC To LSF object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an LPCToLSF System object C, with the same
property values as H. The clone method creates a new unlocked object.

dsp.LPCToLSF.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                     method

                     The getNumOutputs method returns a positive integer that is the
                     number of outputs from the step method. This value will change if you
                     alter any properties that turn outputs on or off.

dsp.LPCToLSF.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the LPCToLSF System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.LPCToLSF.reset

Purpose   Reset values for overwriting invalid outputs to their initial values

Syntax   reset(H)

Description   reset(H) resets the values for overwriting the invalid outputs to their initial values.

Purpose   Convert LPC coefficients to line spectral frequencies

Syntax   `LSF = step(H,A)`
`[..., STATUS] = step(H,A)`

Description `LSF = step(H,A)` converts the LPC coefficients, `A`, to line spectral frequencies, `LSF`, in the range (0 pi). The System object operates along the columns of the input `A`.

`[..., STATUS] = step(H,A)` also returns the status flag, `STATUS`, indicating if the current output is valid when the `ExceptionOutputPort` property is `true`.

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose	Convert linear prediction coefficients to line spectral pairs
Description	The LPCToLSP object converts linear prediction coefficients to line spectral pairs.

To convert LPC to LSP:

1 Define and set up your LPC to LSP converter. See "Construction" on page 5-670.

2 Call step to convert LPC according to the properties of dsp.LPCToLSP. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.LPCToLSP returns a System object, H, that converts linear prediction coefficients (LPCs) to line spectral pairs (LSPs).

H = dsp.LPCToLSP(*'PropertyName'*,*PropertyValue*,...) returns an LPC to LSF System object, H, with each specified property set to the specified value.

Properties   **NumCoarseGridPoints**

Number of coarse subintervals used for finding roots (LSP values)

Specify the number of coarse subintervals, *n*, used for finding line spectral pairs (LSP) values, as a positive scalar integer. LSPs, which are the roots of two particular polynomials related to the input LPC polynomial, always lie in the range (–1, 1). The System object finds these roots using the Chebyshev polynomial root finding method. The object divides the interval (–1, 1) into *n* subintervals and looks for roots in each subinterval. If *n* is set to too small a number in relation to the LPC polynomial order, the object can fail to find some of the roots. The default is 64. This property is tunable.

NumBisects

Value of bisection refinement used for finding roots

Specify the root bisection refinement value, *k*, that the Chebyshev polynomial uses in the root finding method. For each line spectral pair (LSP) the output is within

$$\frac{1}{n \cdot 2^k}$$

of the actual LSP value. Here *n* is the value of the NumCoarseGridPoints property and the object searches a maximum of

$$k \cdot (n\text{-}1)$$

points for finding the roots. The NumBisects property value *k*, must be a positive scalar integer. The default is 4. This property is tunable.

ExceptionOutputPort

Produces output with validity status of LSP output

Set this property to true to return a second output that indicates whether the computed LSP values are valid. The object outputs a vector length equal to the number of channels. A logical value of 1 indicates the output is valid. A logical value of 0 indicates the output is invalid. The LSP outputs are invalid when the object fails to find all the LSP values or when the input LPCs are unstable. The default is false.

OverwriteInvalidOutput

Enable overwriting invalid output with previous output

Specify the action that the object takes for invalid LSP outputs. When you set this property to true, the object overwrites the invalid output with the previous output. When you set this property to false, the object takes no action on invalid outputs and ignores the outputs.

FirstOutputValuesSource

Source of values for first output when output is invalid

Specify the source of values for the first output when the output is invalid as Auto or Property. This property applies only when you set the OverwriteInvalidOutput property to true. The default is Auto. When this property is Auto, the object uses a default value for the first output. The default value corresponds to the LSP representation of an allpass filter.

FirstOutputValues

Value of first output

Specify a numeric vector of LSP values for overwriting an invalid first output. The length of this vector must be one less than the length of the input LPC vector. For multichannel inputs, set this property can to a matrix with the same number of channels as the input or one vector that you apply to every channel. The default is an empty vector. This property applies only when you set the OverwriteInvalidOutput property to true and the FirstOutputValuesSource property to Property.

NonUnityFirstCoefficientAction

First coefficient nonunity action

Specify the action that the object takes when the first coefficient of each channel of the LPC input is not equal to 1. Specify as one of Replace with 1 or Normalize. The default is Replace with 1.

Methods		
	clone	Create LPC To LSP object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
reset	Reset values for overwriting invalid outputs to their initial values
step	Convert linear prediction coefficients to line spectral pairs

Examples　　Convert the LPC coefficients to LSP coefficients:

```
a = [1.0000 0.6149 0.9899 0.0000 0.0031 -0.0082]';
hlpc2lsp = dsp.LPCToLSP;
y = step(hlpc2lsp, a); % Convert to LSP coefficients
```

Algorithms　　This object implements the algorithm, inputs, and outputs described on the LPC to LSF/LSP Conversion block reference page. The object properties correspond to the block parameters, except:

No object property corresponds to the **Output** block parameter. The object only supports LSP outputs.

See Also　　dsp.LSPToLPC | dsp.LPCToLSF

dsp.LPCToLSP.clone

Purpose   Create LPC To LSP object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a LPCToLSP System object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                     the step method

                     The getNumInputs method returns a positive integer that is the number
                     of expected inputs (not counting the object itself) to the step method.
                     This value will change if you alter any properties that turn inputs on or
                     off. You must call the step method with the number of input arguments
                     equal to the result of getNumInputs(*H*).

dsp.LPCToLSP.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the LPCToLSP System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.LPCToLSP.release

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset values for overwriting invalid outputs to their initial values

Syntax   reset(H)

Description reset(H) resets the values for overwriting the invalid outputs to their initial values.

dsp.LPCToLSP.step

Purpose　　　Convert linear prediction coefficients to line spectral pairs

Syntax　　　　```
LSF = step(H,A)
[..., STATUS] = step(H,A)
```

Description　　LSF = step(H,A) converts the LPC coefficients, A , to line spectral
pairs normalized in the range (–1 1), LSP. The object operates along
the columns of the input A.

[..., STATUS] = step(H,A) also returns the status flag, STATUS,
indicating if the current output is valid when the ExceptionOutputPort
property is true.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

Purpose   Convert linear prediction coefficients to reflection coefficients

Description   The LPCToRC object converts linear prediction coefficients to reflection coefficients.

To convert LPC to reflection coefficients:

1 Define and set up your LPC to RC converter. See "Construction" on page 5-681.

2 Call step to convert LPC according to the properties of dsp.LPCToRC. The behavior of step is specific to each object in the toolbox.

Construction H = dsp.LPCToRC returns an LPC to RC System object, H, that converts linear prediction coefficients (LPC) to reflection coefficients (RC).

H = dsp.LPCToRC('*PropertyName*',*PropertyValue*,...) returns an LPC to RC conversion object, H, with each specified property set to the specified value.

Properties   **PredictionErrorOutputPort**

Enable normalized prediction error power output

Set this property to true to return the normalized error power as a vector with one element per input channel. Each element varies between zero and one. The default is true.

ExceptionOutputPort

Produces output with stability status of filter represented by LPC coefficients

Set this property to true to return the stability status of the filter. A logical value of 1 indicate a stable filter. A logical value of 0 indicate an unstable filter. The default is false.

NonUnityFirstCoefficientAction

Action to take when first LPC coefficient is not 1

Specify the action that the object takes when the first coefficient of each channel of the LPC input is not 1. Select `Replace with 1` or `Normalize`. The default is `Replace with 1`.

Methods	clone	Create LPC to RC object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	step	Convert columns of linear prediction coefficients to reflection coefficients

Examples   Convert the linear prediction coefficients to reflection coefficients:

```
load mtlb

hlevinson = dsp.LevinsonSolver;
hlevinson.AOutputPort = true;
hlevinson.KOutputPort = false;

hac = dsp.Autocorrelator;
hlpc2rc = dsp.LPCToRC;
hac.MaximumLagSource = 'Property';

% Compute autocorrelation for lags between [0:10]
hac.MaximumLag = 10;
a = step(hac, mtlb);
A = step(hlevinson, a); % Compute LPC coefficients
```

```
[K, P] = step(hlpc2rc, A); % Convert to RC
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the LPC to/from RC block reference page. The object properties correspond to the block parameters, except:

- There is no object property that corresponds to the **Type of conversion** block parameter. The object always converts LPC to RC.

- The NonUnityFirstCoefficientAction object property corresponds to the **If first input value is not 1** block parameter. There is neither a Normalize and warn nor an Error option for the object.

See Also

dsp.RCToLPC | dsp.LPCToAutocorrelation

dsp.LPCToRC.clone

Purpose   Create LPC to RC object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a LPCToRC System object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.LPCToRC.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the LPCToRC System object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.LPCToRC.release

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Convert columns of linear prediction coefficients to reflection coefficients

Syntax   [K,P] = step(H,A)
                   K = step(H,A)
                   [..., S] = step(H,A)

Description   [K,P] = step(H,A) converts the columns of linear prediction
                   coefficients, A, to reflection coefficients K and outputs the normalized
                   prediction error power, P.

                   K = step(H,A) when you set the PredictionErrorOutputPort
                   property to false, converts the columns of linear prediction coefficients,
                   A, to reflection coefficients K.

                   [..., S] = step(H,A) also outputs the LPC filter stability, S, when
                   you set the ExceptionOutputPort property to true.

                   ───

                   Note The object performs an initialization the first time the step
                   method is executed. This initialization locks nontunable properties and
                   input specifications, such as dimensions, complexity, and data type
                   of the input data. If you change a nontunable property or an input
                   specification, the System object issues an error. To change nontunable
                   properties or inputs, you must first call the release method to unlock
                   the object.

                   ───

dsp.LSFToLPC

Purpose   Convert line spectral frequencies to linear prediction coefficients

Description   The LSFToLPC object converts line spectral frequencies to linear
                   prediction coefficients.

                   To convert LSF to LPC:

                   1 Define and set up your LSF to LPC converter. See "Construction"
                   on page 5-690.

                   2 Call step to convert LSF according to the propeties of dsp.LSFToLPC.
                   The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.LSFToLPC returns an LSF to LPC System object, H, which
                   converts line spectral frequencies (LSFs) to linear prediction coefficients
                   (LPCs).

Methods

clone	Create LSF to LPC object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Convert input line spectral frequencies to linear prediction coefficients

Examples   Convert line spectral frequencies to linear prediction coefficients:

```
a = [1.0000 0.6149 0.9899 0.0000 0.0031 -0.0082]'
```

```
hlpc2lsf = dsp.LPCToLSF;
ylsf = step(hlpc2lsf, a);
hlsf2lpc = dsp.LSFToLPC;
ylpc = step(hlsf2lpc, ylsf) % Check values are same as a.
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the LSF/LSP to LPC Conversion block reference page. The object properties correspond to the block parameters, except:

The object does not have a property that corresponds to the **Input** block parameter. The object's behavior corresponds to the block's behavior when you set the **Input** parameter to LSF in range (0 pi).

See Also   dsp.LPCToLSF | dsp.LSPToLPC

dsp.LSFToLPC.clone

Purpose   Create LSF to LPC object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an LSFToLPC System object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
the step method

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.LSFToLPC.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the LSFToLPC System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.LSFToLPC.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Convert input line spectral frequencies to linear prediction coefficients

Syntax   A = step(H,LSF)

Description A = step(H,LSF) converts the input line spectral frequencies, (LSF), in
the range (0,pi), LSF , to linear prediction coefficients, A. The input
can be a vector or a matrix, where each column of the matrix is treated
as a separate channel.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.LSPToLPC

Purpose   Convert line spectral pairs to linear prediction coefficients

Description   The LSPToLPC object converts line spectral pairs to linear prediction coefficients.

To convert LSP to LPC:

1 Define and set up your LSP to LPC converter. See "Construction" on page 5-698.

2 Call step to convert LSP according to the properties of dsp.LSPToLPC. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.LSPToLPC returns an LSP to LPC System object, H, which converts line spectral pairs (LSPs) to linear prediction coefficients (LPCs).

Methods

clone	Create LSP to LPC object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Convert input line spectral pairs to linear prediction coefficients

Examples   Convert line spectral pairs to linear prediction coefficients:

```
ylsp = [0.7080 0.0103 -0.3021 -0.3218 -0.7093]';
hlsp2lpc = dsp.LSPToLPC;
```

```
ylpc = step(hlsp2lpc, ylsp)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the LSF/LSP to LPC Conversion block reference page. The object properties correspond to the block parameters, except:

No object property corresponds to the **Input** block parameter. The object converts LSP in the range (–1, 1) to LPC.

See Also   dsp.LPCToLSP | dsp.LSFToLPC

dsp.LSPToLPC.clone

Purpose   Create LSP to LPC object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an LSPToLPC System object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.LSPToLPC.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the LSPToLPC System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.LSPToLPC.release

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Convert input line spectral pairs to linear prediction coefficients

Syntax   A = step(H,LSP)

Description A = step(H,LSP) converts the input line spectral pairs in the range
(–1,1), LSP, to linear prediction coefficients, A. The input can be a vector
or a matrix, where each column of the matrix is treated as a separate
channel.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.LUFactor

Purpose   Factor square matrix into lower and upper triangular matrices

Description   The LUFactor object factors a square matrix into lower and upper triangular matrices.

To factor a square matrix into lower and upper triangular matrices:

1 Define and set up your System object. See "Construction" on page 5-706.

2 Call step to factor the square matrix according to the properties of dsp.LUFactor. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.LUFactor returns an LUFactor System object, H, which factors a row permutation of a square input matrix A as $A_p = L \cdot U$, where L is the unit-lower triangular matrix, and U is the upper triangular matrix. The row-pivoted matrix A_p contains the rows of *A* permuted as indicated by the permutation index vector *P*. The equivalent MATLAB code is Ap = A(P,:).

H = dsp.LUFactor(*'PropertyName'*,*PropertyValue*,...) returns an LUFactor object, H, with each specified property set to the specified value.

Properties   **ExceptionOutputPort**

Set to true to output singularity of input

Set this property to true to output the singularity of the input as logical data type values of true or false. An output of true indicates that the current input is singular, and an output of false indicates the current input is nonsingular.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as
|Ceiling|Convergent|Floor|Nearest |Round | Simplest |
Zero|. The default is Floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as Wrap or Saturate. The default is
Wrap.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as Full precision,
Same as input or Custom. The default is Full precision.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype
object with a Signedness of Auto. This property applies when
you set the ProductDataType property to Custom. The default is
numerictype([],32,30).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as Full precision,
Same as input, Same as product or Custom. The default is Full
precision.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype
object with a Signedness of Auto. This property applies when you
set the AccumulatorDataType property to Custom. The default is
numerictype([],32,30).

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

Methods

clone	Create LU Factor object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Decompose matrix into lower and upper triangular matrices

Examples

Decompose a square matrix into the lower and upper components:

```
hlu = dsp.LUFactor;
x = rand(4)
[LU, P] = step(hlu, x);
L = tril(LU,-1)+diag(ones(size(LU,1),1));
U = triu(LU);
y = L*U
```

```
                   % Check back whether this equals the permuted x
                   xp = x(P,:)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the LU Factorization block reference page. The object properties correspond to the block parameters.

See Also   dsp.LDLFactor

Purpose   Create LU Factor object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an LUFactor object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, of
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.LUFactor.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the LUFactor object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.LUFactor.release

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose	Decompose matrix into lower and upper triangular matrices
Syntax	`[LU,P] = step(H,A)` `[LU,P,S] = step(H,A)`
Description	`[LU,P] = step(H,A)` decomposes the matrix A into lower and upper triangular matrices. The output LU is a composite matrix with lower triangle elements from L and upper triangle elements from U . The permutation vector P is the second output. `[LU,P,S] = step(H,A)` returns an additional output S indicating if the input is singular when the `ExceptionOutputPort` property is set to `true`.

> **Note** The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

.

dsp.MatFileReader

Purpose   Read MAT file

Description   The `MatFileReader` object reads V7.3 MAT files.

To read V7.3 MAT files:

1 Define and set up your System object. See "Construction" on page 5-716.

2 Call `step` to read the MAT file according to the properties of `dsp.MatFileReader`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.MatFileReader` returns a System object, H, to read data from a V7.3 MAT file.

`H = dsp.MatFileReader('PropertyName',PropertyValue,...)` returns a MAT file reader System object, H, with each specified property set to the specified value.

`H = dsp.MatFileReader(FILENAME,VARIABLENAME'PropertyName',PropertyValue,...)` returns a MAT file reader System object, H, with the `Filename` property set to *FILENAME*, the `VariableName` property set to *VARIABLENAME*, and other specified properties set to the specified values.

Properties   **Filename**

Name of MAT file from which to read

Specify the name of a MAT file as a string. Specify the full path for the file only if the file is not on the MATLAB path. The default file name is `Untitled.mat`.

VariableName

Name of the variable to read

Name of the variable stored in and read from the MAT file. The default variable name is `x`.

FrameBasedProcessing

Enable frame-based processing

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. The default is `true`.

SamplesPerFrame

Number of samples per output frame

Specify the number of elements to read from the MAT file each time the `step` method is called. This property applies when `FrameBaseProcessing` is `true`. The default number of samples per frame is `1`.

Methods

clone	Create MAT file reader object with same property values
isDone	End-of-file status
isLocked	Locked status (logical) for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of multimedia file reader to read from beginning of file
step	Read data from a variable in the MAT file

Examples

Use `dsp.MatFileReader` and `dsp.MatFileWriter` to stream data.

```
filename = [tempname '.mat'];  % Create variable name

originalData = rand(40,2);
```

```
save(filename,'originalData','-v7.3'); % Write to MAT file

H = dsp.MatFileReader(filename,'VariableName',...
    'originalData','SamplesPerFrame', 4);
while ~isDone(H)          % Stream data into MATLAB
    finalData = step(H);
end
```

See Also   dsp.MatFileWriter

Purpose   Create MAT file reader object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a MatFileReader System object C, with the same
property values as H.

The clone method creates a new unlocked object with uninitialized
states.

dsp.MatFileReader.isLocked

Purpose   Locked status (logical) for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the MatFileReader object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   End-of-file status

Syntax   STATUS = isDone(H)

Description   STATUS = isDone(H) returns a logical value, STATUS. When the
                    MatFileReader object, H , reaches the end of the MAT file, STATUS is
                    true.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections. This method lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset internal states of multimedia file reader to read from beginning of file

Syntax   reset(H)

Description   reset(H) resets the MatFileReader object to read from the beginning of the file.

dsp.MatFileReader.step

Purpose   Read data from a variable in the MAT file

Syntax   X = step(H)

Description X = step(H) reads data, X, from a variable stored in a MAT-file. The variable is assumed to be *N*-dimensional and a MATLAB built-in datatype. If the FrameBasedProcessing property is true, the data is read into MATLAB by reading along the first dimension. If FrameBasedProcessing is false, the data is read into MATLAB along the last dimension.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Write MAT file

Description   The MatFileWriter object writes data to a V7.3 MAT file.

To write data to a V7.3 MAT file:

1 Define and set up your System object. See "Construction" on page 5-725.

2 Call step to write data according to the properties of dsp.MatFileWriter. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.MatFileWriter returns a MAT file writer System object, H, that writes data to a V7.3 MAT file.

H = dsp.MatFileWriter(*'PropertyName'*,*PropertyValue*,...) returns a MAT file writer System object, H, with each specified property set to the specified value.

H = dsp.MatFileWriter(*FILENAME*,*'PropertyName'*,*PropertyValue*,...) returns a MAT file writer System object, H, with Filename property set to *FILENAME* and other specified properties set to the specified values.

Properties   **Filename**

Name of MAT file to write

Specify the name of a MAT file as a string. Specify the full path for the file only if the file is not on the MATLAB path. The default file name is Untitled.mat.

VariableName

Name of the variable to write

Name of the variable to which to write. This variable is stored in the MAT file. The default variable name is x.

FrameBasedProcessing

dsp.MatFileWriter

Enable frame-based processing

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. In frame-based mode, this System object streams the data to the first dimension. In sample-based mode, a new dimension is appended to the end of the data set. The default is `true`.

Methods

clone	Create MAT file writer object with the same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Write one frame of MAT file data

Examples   Use `dsp.MatFileWriter` and `dsp.MatFileReader` to stream data.

```
% First, create a variable name
      filename = [tempname '.mat'];

% Next, write that variable to a MAT-file.
      hmfw = dsp.MatFileWriter(filename, 'VariableName', 'originalData')
      for i=1:10
          originalData = rand(4,2);
          step(hmfw, originalData);
      end
      release(hmfw); % This will close the MAT file

% Finally, load the variable back into MATLAB.
      data = load(filename, 'originalData');
```

See Also   dsp.MatFileReader

Purpose   Create MAT file writer object with the same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a MatFileWriter System object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the MatFileWriter System
object H.

The isLocked method returns a logical value that indicates whether
input attributes and nontunable properties for the object are locked. The
object performs an internal initialization the first time the step method
is executed. This initialization locks nontunable properties and input
specifications, such as dimensions, complexity, and data type of the
input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections. `release` also closes the file. This method lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Write one frame of MAT file data

Syntax   step(H,X)

Description   step(H,X) writes one frame of data, X , to the variable stored in the
                     MAT file. The variable is assumed to be N-dimensional and a MATLAB
                     built-in data type. If the FrameBasedProcessing property is true, the
                     data is written to the file by concatenating along the first dimension. If
                     FrameBasedProcessing is false, the data is written by appending an
                     additional dimension to the end of the data set.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

Purpose   Find maximum value of input or sequence of inputs

Description   The Maximum object finds the maximum values of an input or sequence of inputs.

To compute the maximum value of an input or sequence of inputs:

1 Define and set up your System object. See "Construction" on page 5-731.

2 Call step to find the maximum according to the properties of dsp.Maximum. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.Maximum returns an object, H, that computes the value and index of the maximum elements in an input or a sequence of inputs along the specified Dimension.

H = dsp.Maximum('*PropertyName*',*PropertyValue*,...) returns a maximum-finding object, H, with each specified property set to the specified value.

Properties   **ValueOutputPort**

Output maximum value

Set this property to true in order to output the maximum of the input. This property applies only when you set the RunningMaximum property to false. The default is true.

RunningMaximum

Calculate over single input or multiple inputs

When you set this property to true, the object computes the maximum value over successive calls to the step method. When you set this property to false, the object computes the maximum value over the current input. The default is false.

IndexOutputPort

Output index of maximum value

Set this property to `true` to output the index of the maximum value of the input. This property applies only when you set the `RunningMaximum` property to `false`. The default is `true`.

ResetInputPort

Additional input to enable resetting of running maximum

Set this property to `true` to enable resetting the running maximum. When you set this property to `true`, you must specify a reset input to the `step` method to reset the running maximum. This property applies only when you set the `RunningMaximum` property to `true`. The default is `false`.

ResetCondition

Condition that triggers resetting of running maximum

Specify the event that resets the running maximum as one of | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero` |. This property applies only when you set the `ResetInputPort` property to `true`. The default is `Non-zero`.

IndexBase

Numbering base for index of maximum value

Specify whether to start the index numbering from `One` or `Zero` when computing the index of the maximum value. This property applies only when you set the `IndexOutputPort` property to `true`. The default is `One`.

Dimension

Dimension to operate along

Specify how the maximum calculation is performed over the data as one of | `All` | `Row` | `Column` | `Custom` |. This property applies when you set the `RunningMaximum` property to `false`. The default is `Column`.

CustomDimension

Numerical dimension to calculate over

Specify the integer dimension of the input signal over which the object finds the maximum. The cannot exceed the number of dimensions in the input signal. This property only applies when you set the `Dimension` property to `Custom`. The default is `1`.

ROIProcessing

Enable region-of-interest processing

Set this property to `true` to enable calculation of the maximum value within a particular region of an image. This property applies when you set the `Dimension` property to `All` and the `RunningMaximum` property to `false`. The default is `false`.

For full ROI processing support, install the Computer Vision System Toolbox product. If you only have the DSP System Toolbox product installed, you can only specify the value of the `ROIForm` property as `Rectangles`.

ROIForm

Type of region of interest

Specify the type of region of interest as one of | `Rectangles` | `Lines` | `Label matrix` | `Binary mask` |. This property applies only when you set the `ROIProcessing` property to `true`. The default is `Rectangles`.

For full ROI processing support, install the Computer Vision System Toolbox product. If you have only the DSP System Toolbox product installed, you can only specify the as `Rectangles`.

ROIPortion

Calculate over entire ROI or just perimeter

Specify whether to calculate the maximum over the `Entire ROI` or the `ROI perimeter`. This property applies only when you set the `ROIForm` property to `Rectangles`. The default is `Entire ROI`.

ROIStatistics

Calculate statistics for each ROI or one for all ROIs

Specify whether to calculate `Individual statistics for each ROI` or a `Single statistic for all ROIs`. This property applies only when you set the `ROIForm` property to `Rectangles`, `Lines`, or `Label matrix`.

ValidityOutputPort

Output flag indicating if any part of ROI is outside input image

When you set the `ROIForm` property to one of | `Lines` | `Rectangles` |, set this property to `true` to return the validity of the specified ROI being completely inside of the image. When you set the `ROIForm` property to `Label Matrix`, set this property to `true` to return the validity of the specified label numbers. The default is `false`.

FrameBasedProcessing

Process input as frames or samples

Set this property to `true` to enable frame-based processing for 2-D inputs. Set this property to `false` to enable sample-based processing. The object always performs sample-based processing for N-D inputs where N is greater than 2. This property applies when you set the `RunningMaximum` to `true`. The default is `true`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`.

OverflowAction

Action to take when integer input is out-of-range

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap.

ProductDataType

Data type of product

Specify the product fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the AccumulatorDataType property to Custom. The default is numerictype([],32,30).

AccumulatorDataType

Data type of accumulator

Specify the accumulator fixed-point data type as one of | Same as product | Same as input | Custom |. The default is Same as product.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the AccumulatorDataType property to Custom. The default is numerictype([],32,30).

Methods

clone	Create maximum-finding object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset computation of running maximum
step	Maximum value

Examples

Find a maximum value and its index:

```
hmax1 = dsp.Maximum;
x = randn(100,1);
[y, I] = step(hmax1, x);
```

Compute a running maximum:

```
hmax2 = dsp.Maximum;
hmax2.RunningMaximum = true;
x = randn(100,1);
y = step(hmax2, x);
% y(i) is the maximum of all values in the vector x(1:i)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Maximum block reference page. The object properties correspond to the block parameters, except:

Treat sample-based row input as a column block parameter is not supported by the dsp.Maximum object.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the FrameBasedProcessing property. The block uses the **Input**

processing parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.Minimum | dsp.Mean

dsp.Maximum.clone

Purpose   Create maximum-finding object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a Maximum object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.Maximum.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step
                 method.

                 The getNumOutputs method returns a positive integer that is the
                 number of outputs from the step method. This value will change if you
                 alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the Maximum object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.Maximum.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset computation of running maximum

Syntax   reset(H)

Description   reset(H) resets the computation of the running maximum for the
Maximum object H.

dsp.Maximum.step

Purpose   Maximum value

Syntax   [VAL,IND] = step(H,X)
                 VAL = step(H,X)
                 IND = step(H,X)
                 VAL = step(H,X,R)
                 [...] = step(H,I,ROI)
                 [...] = step(H,I,LABEL,LABELNUMBERS)
                 [...,FLAG] = step(H,I,ROI)
                 [...,FLAG] = step(H,I,LABEL,LABELNUMBERS)

Description [VAL,IND] = step(H,X) returns the maximum value, VAL, and the
                 index or position of the maximum value, IND, along the specified
                 Dimension of X.

                 VAL = step(H,X) returns the maximum value, VAL, of the input X.
                 When the RunningMaximum property is true, VAL corresponds to the
                 maximum value over successive calls to the step method.

                 IND = step(H,X) returns the zero- or one-based index IND of
                 the maximum value. To enable this type of processing, set the
                 IndexOutputPort property to true and the ValueOutputPort and
                 RunningMaximum properties to false.

                 VAL = step(H,X,R) resets the state of H based on the value of reset
                 signal, R, and the ResetCondition property. To enable this type
                 of processing, set the RunningMaximum property to true and the
                 ResetInputPort property to true.

                 [...] = step(H,I,ROI) computes the maximum of an input image,
                 I, within the given region of interest, ROI. To enable this type of
                 processing, set the ROIProcessing property to true and the ROIForm
                 property to Lines, Rectangles or Binary mask.

                 [...] = step(H,I,LABEL,LABELNUMBERS) computes the maximum of
                 an input image, I, for a region whose labels are specified in the vector
                 LABELNUMBERS. To enable this type of processing, set the ROIProcessing
                 property to true and the ROIForm property to Label matrix.

`[...,FLAG] = step(H,I,ROI)` returns `FLAG`, indicating whether the given region of interest is within the image bounds. To enable this type of processing, set the `ROIProcessing` and `ValidityOutputPort` properties to `true` and the `ROIForm` property to `Lines`, `Rectangles` or `Binary mask`.

`[...,FLAG] = step(H,I,LABEL,LABELNUMBERS)` returns `FLAG`, indicating whether the input label numbers are valid. To enable this type of processing, set the `ROIProcessing` and `ValidityOutputPort` properties to `true` and the `ROIForm` property to `Label matrix`.

For full ROI processing support, install the Computer Vision System Toolbox product. If you only have the DSP System Toolbox product installed, you can only specify the value of the `ROIForm` property as `Rectangles`.

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.Mean

Purpose	Find mean value of input or sequence of inputs

Description

The Mean object finds the mean of an input or sequence of inputs.

To compute the mean of an input or sequence of inputs:

1 Define and set up your System object. See "Construction" on page 5-746.

2 Call step to compute the mean according to the properties of dsp.Mean. The behavior of step is specific to each object in the toolbox.

Construction

H = dsp.Mean returns an object, H, that computes the mean of an input or a sequence of inputs.

H = dsp.Mean(*'PropertyName'*,*PropertyValue*,...) returns a mean-finding object, H, with each specified property set to the specified value.

Properties

RunningMean

Calculate over single input or multiple inputs

When you set this property to true, the object calculates the mean over successive calls to the step method. When you set this property to false, the object computes the mean over the current input. The default is false.

ResetInputPort

Additional input to enable resetting of running mean

Set this property to true to enable resetting of the running mean. When you set this property to true, you must specify a reset input to the step method to reset the running mean. This property applies only when you set the RunningMean property to true. The default is false.

ResetCondition

Condition that triggers resetting of running mean

Specify the event that resets the running maximum as one of | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero` |. This property applies only when you set the `ResetInputPort` property to `true`. The default is `Non-zero`.

Dimension

Dimension to operate along

Specify how the mean calculation is performed over the data as one of | `All` | `Row` | `Column` | `Custom` |. This property applies when you set the `RunningMean` property to `false`. The default is `Column`.

CustomDimension

Numerical dimension to calculate over

Specify the integer dimension, indexed from one, of the input signal over which the object calculates the mean. The value cannot exceed the number of dimensions in the input signal. This property only applies when you set the `Dimension` property to `Custom`. The default is `1`.

ROIProcessing

Enable region-of-interest processing

Set this property to `true` to enable calculation of the mean within a particular region of an image. This property applies when you set the `Dimension` property to `All` and the `RunningMean` property to `false`. The default is `false`.

For full ROI processing support, install the Computer Vision System Toolbox product. If you only have the DSP System Toolbox product installed, you can only specify the value of the `ROIForm` property as `Rectangles`.

ROIForm

Type of region of interest

Specify the type of region of interest as one of | `Rectangles` | `Lines` | `Label matrix` |. This property applies only when you set the `ROIProcessing` property to `true`. The default is `Rectangles`.

For full ROI processing support, install the Computer Vision System Toolbox product. If you have only the DSP System Toolbox product installed, you can only specify the as `Rectangles`.

ROIPortion

Calculate over entire ROI or just perimeter

Specify whether to calculate the mean over the `Entire ROI` or the `ROI perimeter`. This property applies only when you set the `ROIForm` property to `Rectangles`. The default is `Entire ROI`.

ROIStatistics

Calculate statistics for each ROI or one for all ROIs

Specify whether to calculate `Individual statistics for each ROI` or a `Single statistic for all ROIs`. This property applies only when you set the `ROIForm` property to `Rectangles`, `Lines`, or `Label matrix`. The default is `Individual statistics for each ROI`.

ValidityOutputPort

Output flag indicating if any part of ROI is outside input image

When you set the `ROIForm` property to one of | `Lines` | `Rectangles` | `Binary mask` |, set this property to `true` to return the validity of the specified ROI being completely inside of the image. When you set the `ROIForm` property to `Label Matrix`, set this property to `true` to return the validity of the specified label numbers. The default is `false`.

FrameBasedProcessing

Process input as frames or samples

Set this property to `true` to enable frame-based processing for 2-D inputs. Set this property to `false` to enable sample-based

processing. The object always performs sample-based processing for N-D inputs where N is greater than 2. This property applies when you set the RunningMean to true. The default is true.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor.

OverflowAction

Action to take when integer input is out-of-range

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap.

AccumulatorDataType

Data type of accumulator

Specify the accumulator fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the AccumulatorDataType property to Custom. The default is numerictype([],32,30).

OutputDataType

Data type of output

Specify the output fixed-point data type as one of | Same as accumulator | Same as input | Custom |. The default is Same as accumulator.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],32,30)`.

Methods

clone	Create mean object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of mean-finding object
step	Mean value

Examples

Compute the mean of a signal:

```
hmean1 = dsp.Mean;
x = randn(100,1);
y = step(hmean1, x);
```

Compute the running mean of a signal:

```
hmean2 = dsp.Mean;
hmean2.RunningMean = true;
x = randn(100,1);
```

```
 y = step(hmean2, x);
% y(i) is the mean of all values in the vector x(1:i)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Mean block reference page. The object properties correspond to the block parameters, except:

> **Treat sample-based row input as a column** block parameter is not supported by the dsp.Mean object.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the FrameBasedProcessing property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.Maximum | dsp.Minimum

dsp.Mean.clone

Purpose   Create mean object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a Mean object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

Purpose	Number of outputs of step method
Syntax	N = getNumOutputs(H)
Description	N = getNumOutputs(H) returns the number of outputs, N, of the step method.
	The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the Mean object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.Mean.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset internal states of mean-finding object

Syntax   reset(H)

Description   reset(H) sets the internal states of the Mean object H to their initial
                  values.

dsp.Mean.step

Purpose　　Mean value

Syntax　　　Y = step(H,X)
　　　　　　　Y = step(H,X,R)
　　　　　　　Y = step(H,X,ROI)
　　　　　　　Y = step(H,X,LABEL,LABELNUMBERS)
　　　　　　　[Y,FLAG] = step(H,X,ROI)
　　　　　　　[Y,FLAG] = step(H,X,LABEL,LABELNUMBERS)

Description　Y = step(H,X) computes the mean of X. When you set the RunningMean property to true, Y corresponds to the mean successive calls to the step method.

Y = step(H,X,R) resets the computation of the running mean based on the value of the reset signal, R, and the ResetCondition property. To enable this type of processing, set the RunningMean property to true and the ResetInputPort property to true.

Y = step(H,X,ROI) computes the mean of input image X within the given region of interest ROI. To enable this type of processing, set the ROIProcessing property to true and the ROIForm property to Lines, Rectangles or Binary mask.

Y = step(H,X,LABEL,LABELNUMBERS) computes the mean of the input image, X, for the region whose labels are specified in the vector LABELNUMBERS. The regions are defined and labeled in the matrix LABEL. To enable this type of processing, set the ROIProcessing property to true and the ROIForm property to Label matrix.

[Y,FLAG] = step(H,X,ROI) returns FLAG, indicating whether the given region of interest ROI, is within the image bounds. To enable this type of processing, set the ROIProcessing and ValidityOutputPort properties to true and the ROIForm property to Lines, Rectangles or Binary mask.

[Y,FLAG] = step(H,X,LABEL,LABELNUMBERS) returns FLAG which indicates whether the input label numbers are valid. To enable this type of processing, set the ROIProcessing and ValidityOutputPort properties to true and the ROIForm property to Label matrix.

For full ROI processing support, install the Computer Vision System Toolbox product. If you have only the DSP System Toolbox product installed, you can only specify the value of this property as `Rectangles`.

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.Median

Purpose　　　Median value of input

Description　　The Median object computes the median value of the input. The object can compute the median along each dimension (row or column) of the input or of the entire input.

To compute the median of the input:

1 Define and set up your median System object. See "Construction" on page 5-760.

2 Call step to compute the median according to the properties of dsp.Median. The behavior of step is specific to each object in the toolbox.

Construction　　H = dsp.Median returns a median System object, H, that computes the median along the columns of the input using the Quick sort sorting method.

H = dsp.Median('*PropertyName*',*PropertyValue*,...) returns a median System object, H, with each property set to the value you specify.

Properties　　**SortMethod**

Sort method

Specify the method the object should use to sort the data before computing the median. You can specify Quick sort or Insertion sort. The quick sort algorithm uses a recursive sort method and is faster at sorting more than 32 elements. The insertion sort algorithm uses a nonrecursive method and is faster at sorting less than 32 elements. If you are using the Median object to generate code, you should use the insertion sort algorithm to prevent recursive function calls in your generated code. The default is Quick sort.

Dimension

Dimension to operate along

Specify the dimension along which the object computes the median values. You can specify one of | All | Row | Column | Custom |. The default is Column.

CustomDimension

Numerical dimension to operate along

Specify the dimension of the input signal (as a one-based value), over which the object computes the median. The cannot exceed the number of dimensions in the input signal. This property applies only when you set the Dimension property to Custom. The default is 1.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap.

ProductDataType

Product word and fraction lengths

Specify the product data type as one of | Same as input | Custom |. The default is Same as input.

CustomProductDataType

Product word and fraction lengths

Specify the product data type as a scaled numerictype object with a Signedness of Auto. This property applies only when

you set the ProductDataType property to Custom. The default is numerictype([],32,30).

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator data type as one of | Same as product | Same as input | Custom |. The default is Same as product.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the fixed-point accumulator data type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the AccumulatorDataType property to Custom. The default is numerictype([],32,30).

OutputDataType

Output word and fraction lengths

Specify the output data type as one of | Same as accumulator | Same as product | Same as input | Custom |. The default is Same as accumulator.

CustomOutputDataType

Output word and fraction lengths

Specify the data type of the output as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the OutputDataType property to Custom. The default is numerictype([],16,15).

Methods	clone	Create median object with same property values
	getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Median value

Examples   Compute the median value of the input column:

```
h = dsp.Median;
x = [7 -9 0 -1 2 0 3 5 -9]';
y = step(h, x)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Median block reference page. The object properties correspond to the block properties, except:

> **Treat sample-based row input as a column** block parameter is not supported by the dsp.Median System object.

See Also   dsp.Mean | dsp.Minimum | dsp.Maximum | dsp.Variance

dsp.Median.clone

Purpose Create median object with same property values

Syntax C = clone(H)

Description C = clone(H) creates a median object C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.Median.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                     method.

                     The getNumOutputs method returns a positive integer that is the
                     number of outputs from the step method. This value will change if you
                     alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the Median System object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.Median.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Median value

Syntax   Y = step(H,X)

Description   Y = step(H,X) computes the median value of the input X and returns
                   the result in Y.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

Purpose	Find minimum values of input or sequence of inputs
Description	The Minimum object finds the minimum value of an input or sequence of inputs.

To compute the minimum value of an input or sequence of inputs:

1 Define and set up your System object. See "Construction" on page 5-770.

2 Call step to compute the minimum according to the properties of dsp.Minimum. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.Minimum returns an object, H, that computes the value and/or index of the minimum elements in an input or a sequence of inputs over the specified Dimension.

H = dsp.Minimum('*PropertyName*',*PropertyValue*,...) returns a minimum-finding object, H, with each specified property set to the specified value.

Properties   **ValueOutputPort**

Output minimum value

Set this property to true in order to output the minimum value of the input. This property applies only when you set theRunningMinimum property to false. The default is true.

RunningMinimum

Calculate over single input or multiple inputs

When you set this property to true, the object computes the minimum value over successive calls to the step method. When you set this property to false, the object computes the minimum value over the current input. The default is false.

IndexOutputPort

Output index of minimum value

Set this property to `true` to output the index of the minimum value of the input. This property applies only when you set the `RunningMinimum` property to `false`. The default is `true`.

ResetInputPort

Additional input to enable resetting of running minimum

Set this property to `true` to enable resetting of the running minimum. When you set this property to `true`, you must specify a reset input to the `step` method to reset the running minimum. This property applies only when you set the `RunningMinimum` property to `true`. The default is `false`.

ResetCondition

Condition that triggers resetting of running minimum

Specify the event that resets the running minimum as one of | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero` |. This property applies only when you set the `ResetInputPort` property to `true`. The default is `Non-zero`.

IndexBase

Numbering base for index of minimum value

Specify the numbering used when computing the index of the minimum value as starting from either `One` or `Zero`. This property applies only when you set the `IndexOutputPort` property to `true`. The default is `One`.

Dimension

Dimension to operate along

Specify how the minimum calculation is performed over the data as one of | `All` | `Row` | `Column` | `Custom` |. This property applies when you set the `RunningMinimum` property to `false`. The default is `Column`.

CustomDimension

Numerical dimension to calculate over

Specify the integer dimension of the input signal over which the object finds the minimum. The cannot exceed the number of dimensions in the input signal. This property only applies when you set the `Dimension` property to `Custom`. The default is `1`.

ROIProcessing

Enable region-of-interest processing

Set this property to `true` to enable calculation of the minimum value within a particular region of an image. This property applies when you set the `Dimension` property to `All` and the `RunningMinimum` property to `false`. The default is `false`.

For full ROI processing support, install the Computer Vision System Toolbox product. If you only have the DSP System Toolbox product installed, you can only specify the value of the `ROIForm` property as `Rectangles`.

ROIForm

Type of region of interest

Specify the type of region of interest as one of | `Rectangles` | `Lines` | `Label matrix` | `Binary mask` |. This property applies only when you set the `ROIProcessing` property to `true`. The default is `Rectangles`.

For full ROI processing support, install the Computer Vision System Toolbox product. If you have only the DSP System Toolbox product installed, you can only specify the as `Rectangles`.

ROIPortion

Calculate over entire ROI or just perimeter

Specify whether to calculate the minimum over the `Entire ROI` or the `ROI perimeter`. This property applies only when you set the `ROIForm` property to `Rectangles`. The default is `Entire ROI`.

ROIStatistics

Calculate statistics for each ROI or one for all ROIs

Specify whether to calculate `Individual statistics for each ROI` or a `Single statistic for all ROIs`. This property applies only when you set the `ROIForm` property to `Rectangles`, `Lines`, or `Label matrix`.

ValidityOutputPort

Output flag indicating if any part of ROI is outside input image

When you set the `ROIForm` property to `Lines` or `Rectangles`, set this property to `true` to return the validity of the specified ROI being completely inside of the image. When you set the `ROIForm` property to `Label Matrix`, set this property to `true` to return the validity of the specified label numbers. The default is `false`.

FrameBasedProcessing

Process input as frames or samples

Set this property to `true` to enable frame-based processing for 2-D inputs. Set this property to `false` to enable sample-based processing. The object always performs sample-based processing for N-D inputs where N is greater than 2. This property applies when you set the `RunningMinimum` to `true`. The default is `true`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`.

OverflowAction

Action to take when integer input is out-of-range

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`.

ProductDataType

Data type of product

Specify the product fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the AccumulatorDataType property to Custom. The default is numerictype([],32,30).

AccumulatorDataType

Data type of accumulator

Specify the accumulator fixed-point data type as one of | Same as product | Same as input | Custom |. The default is Same as product.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the AccumulatorDataType property to Custom. The default is numerictype([],32,30).

Methods	clone	Create minimum-finding object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of minimum-finding object
step	Operate on inputs to calculate outputs

Examples   Find a minimum value and its index:

```
hmin1 = dsp.Minimum;
x = randn(100,1);
[y, I] = step(hmin1, x);
```

Compute a running minimum:

```
hmin2 = dsp.Minimum;
hmin2.RunningMinimum = true;
x = randn(100,1);
y = step(hmin2, x);
% y(i) is the minimum of all values in the vector x(1:i)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Minimum block reference page. The object properties correspond to the block parameters, except:

Treat sample-based row input as a column block parameter is not supported by the dsp.Minimum object.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the FrameBasedProcessing property. The block uses the **Input**

dsp.Minimum

processing parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.Maximum | dsp.Mean

Purpose   Create minimum-finding object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a Minimum object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

dsp.Minimum.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
method.

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

dsp.Minimum.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the Minimum object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.Minimum.reset

Purpose   Reset internal states of minimum-finding object

Syntax   reset(H)

Description reset(H) sets the internal states of the Minimum object H to their initial values.

Purpose   Operate on inputs to calculate outputs

Syntax   ```
[VAL,IND] = step(H,X)
VAL = step(H,X)
IND = step(H,X)
VAL = step(H,X,R)
[...] = step(H,I,ROI)
[...] = step(H,I,LABEL,LABELNUMBERS)
[...,FLAG] = step(H,I,ROI)
[...,FLAG] = step(H,I,LABEL,LABELNUMBERS)
```

**Description**   [VAL,IND] = step(H,X) returns the minimum value, VAL, and the
index or position of the minimum value, IND, along the specified
Dimension of X.

VAL = step(H,X) returns the minimum value, VAL, of the input X.
When the RunningMinimum property is true, VAL corresponds to the
minimum value over successive calls to the step method.

IND = step(H,X) returns the zero- or one-based index IND of the
minimum value when the IndexOutputPort property is true and the
ValueOutputPort property is false. You must set the RunningMinimum
property to false to use this syntax.

VAL = step(H,X,R) resets the state of H based on the value of reset
signal, R, and the ResetCondition property. To enable this type
of processing, set the RunningMinimum property to true and the
ResetInputPort property to true.

[...] = step(H,I,ROI) computes the minimum of an input image,
I, within the given region of interest, ROI. To enable this type of
processing, set the ROIProcessing property to true and the ROIForm
property to Lines, Rectangles or Binary mask.

[...] = step(H,I,LABEL,LABELNUMBERS) computes the minimum
of an input image, I, for a region whose labels are specified in the
vector LABELNUMBERS. This property applies only when you set the
ROIProcessing property to true and the ROIForm property to Label
matrix.

[...,FLAG] = step(H,I,ROI) returns FLAG, indicating hether the given region of interest is within the image bounds. To enable this type of processing, set the ROIProcessing and ValidityOutputPort properties to true and the ROIForm property to Lines, Rectangles or Binary mask.

[...,FLAG] = step(H,I,LABEL,LABELNUMBERS) returns FLAG, indicating whether the input label numbers are valid. To enable this type of processing, set the ROIProcessing and ValidityOutputPort properties to true and the ROIForm property to Label matrix.

For full ROI processing support, install the Computer Vision System Toolbox product. If you have only the DSP System Toolbox product installed, you can only specify the value of the ROIForm property as Rectangles.

**Note** The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

**Purpose**          Read video frames, audio samples, or both from multimedia file

---

**Note** `MultimediaFileReader` will be removed in a future release. Use `AudioFileReader` instead. See `dsp.AudioFileReader`.

---

**Description**      The `MultimediaFileReader` object reads video frames, audio samples, or both from a multimedia file.

To read video frames, audio samples, or both from a multimedia file:

**1** Define and set up your multimedia file reader. See "Construction" on page 5-785.

**2** Call `step` to read from a multimedia file according to the properties of `signalblks.MultimediaFileReader`. The behavior of `step` is specific to each object in the toolbox.

**Construction**     `H = signalblks.MultimediaFileReader` returns a multimedia file reader System object, `H`, to read video and/or audio from a multimedia file.

`H = signalblks.MultimediaFileReader('`*PropertyName*`',`*PropertyValue*`,...)` returns a multimedia file reader System object, `H`, with each specified property set to the specified value.

`H = signalblks.MultimediaFileReader(FILENAME,'`*PropertyName*`',`*PropertyValue*`,` returns a multimedia file reader System object, `H`, with `Filename` property set to *FILENAME* and other specified properties set to the specified values.

**Properties**       **Filename**

Name of multimedia file from which to read

Specify the name of a multimedia file as a string. Specify the full path for the file only if the file is not on the MATLAB path. On UNIX platforms, the multimedia file reader only supports uncompressed AVI files. The default is `speech_dft.avi`.

**PlayCount**

Number of times to play file

Specify a positive integer or `inf` to represent the number of times to play the file. The default is `inf`.

**AudioOutputPort**

Choose to output audio data

Use this property to control the audio output from the multimedia file reader. This property applies only when a multimedia file contains both audio and video streams. The default is `true`.

**VideoOutputPort**

Choose to output video data

Use this property to control the video output from the multimedia file reader. This property only applies when a multimedia file contains both audio and video streams. The default is `false`.

**SamplesPerAudioFrame**

Number of samples per audio frame

Specify the number of samples per audio frame as a positive scalar integer value. This property applies when the multimedia file contains only audio data. The default is `1024`.

**ImageColorSpace**

Choose whether output is RGB, intensity, or YCbCr

Specify whether you want the multimedia file reader to output `RGB`, `Intensity`, or `YCbCr 4:2:2` video frames. This property applies only when the multimedia file contains video. The default is `RGB`.

**VideoOutputDataType**

Data type of video data output

Set the data type of the video data output from the multimedia file reader. The multimedia file reader supports the following output data types: `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32` or `Inherit`. This property applies if the multimedia file contains video. The default is `single`.

**AudioOutputDataType**

Data type of audio samples output

Set the data type of the audio data output from the multimedia file reader. The multimedia file reader supports the following output data types: `double`, `single`, `int16`, `uint8`. This property applies only if the multimedia file contains audio. The default is `int16`.

**Methods**

| | |
|---|---|
| clone | Create multimedia file reader object with same property values |
| close | Release resources |
| getNumInputs | Number of expected inputs to step method |
| getNumOutputs | Number of outputs of step method |
| info | Information about specific multimedia file |
| isDone | End-of-file status (logical) |
| isLocked | Locked status for input attributes and nontunable properties |

# signalblks.MultimediaFileReader

| | |
|---|---|
| reset | Reset internal states of multimedia file reader to read from beginning of file |
| step | Output one frame of video frames, audio samples, or both to multimedia signal |

**Examples**

Read an audio file and play the file back using the standard audio output device:

```
hmfr = signalblks.MultimediaFileReader('speech_dft.avi');
hap = dsp.AudioPlayer('SampleRate', 22050);
while ~isDone(hmfr)
audio = step(hmfr);
step(hap, audio);
end
close(hmfr); % close the input file
close(hap); % close the audio output device
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the From Multimedia File block reference page. The object properties correspond to the block parameters, except:

- The object has no corresponding property for the **Inherit sample time from file** block parameter. The object always inherits the sample time from the file.

- The object has no corresponding property for the **Output end-of-file indicator** parameter. The object always outputs EOF as the last output.

- The **Multimedia outputs** block parameter corresponds to both the AudioOutputPort and the VideoOutputPort object properties.

- The object has no corresponding property for the **Image signal** block parameter. The object always outputs an *M*-by-*N*-by-*P* color video signal.

See Also   signalblks.MultimediaFileWriter

signalblks.MultimediaFileReader.clone

Purpose Create multimedia file reader object with same property values

---

Note `MultimediaFileReader` will be removed in a future release. Use `AudioFileReader` instead. See `dsp.AudioFileReader.clone`.

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `MultimediaFileReader` System object `C`, with the same property values as `H`.

The clone method creates a new unlocked object with uninitialized states.

Purpose   Release resources

> **Note** `MultimediaFileReader` will be removed in a future release. Use `AudioFileReader` instead. See `dsp.AudioFileWriter.release` .

Syntax   `close(H)`

Description   `close(H)` releases system resources (such as memory, file handles or hardware connections).

Purpose   Number of expected inputs to step method

> **Note** `MultimediaFileReader` will be removed in a future release. Use
> `AudioFileReader` instead. See `dsp.AudioFileReader.getNumInputs`.

Syntax   `N = getNumInputs(H)`

Description   `N = getNumInputs(H)` returns the number of outputs, N, to the `step`
                     method

                     The `getNumInputs` method returns a positive integer that is the number
                     of expected inputs (not counting the object itself) to the `step` method.
                     This value will change if you alter any properties that turn inputs on or
                     off. You must call the `step` method with the number of input arguments
                     equal to the result of `getNumInputs(H)`.

signalblks.MultimediaFileReader.getNumOutputs

Purpose   Number of outputs of step method

> **Note** MultimediaFileReader will be removed in a future release. Use
> AudioFileReader instead. See dsp.AudioFileReader.getNumOutputs.

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
method

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

signalblks.MultimediaFileReader.info

Purpose   Information about specific multimedia file

> **Note** `MultimediaFileReader` will be removed in a future release. Use
> `AudioFileReader` instead. See `dsp.AudioFileReader.info`.

Syntax   `S = info(H)`

Description `S = info(H)` returns a MATLAB structure, `S`, with information about
the multimedia file specified in the `Filename` property. The number
of fields for `S` varies depending on the audio or video content of the
file. The possible fields and values for the structure `S` are shown in
the following table.

Field	Value
Audio	Logical value indicating if the file has audio content.
Video	Logical value indicating if the file has video content.
AudioSampleRate	Audio sampling rate of the multimedia file in hertz. This field is available when the file has audio content.
AudioNumBits	Number of bits used to encode the audio stream. This field is available when the file has audio content.
AudioNumChannels	Number of audio channels. This field is available when the file has audio content.
FrameRate	Frame rate of the video stream in frames per second. The value may vary from the actual frame rate of the recorded video, and considers any synchronization issues between audio and video streams when the file contains both audio and video content. This variation indicates that video frames may be dropped if the audio stream leads the video stream by more than 1/(actual video frames per second). This field is available when the file has video content.

Field	Value
VideoSize	Video size as a two-element numeric vector of the form [VideoWidthInPixels, VideoHeightInPixels]. This field is available when the file has video content.
VideoFormat	Video signal format. This field is available when the file has video content.

signalblks.MultimediaFileReader.isDone

Purpose   End-of-file status (logical)

Note `MultimediaFileReader` will be removed in a future release. Use
`AudioFileReader` instead. See `dsp.AudioFileReader.isDone`.

Syntax   STATUS = isDone(H)

Description   STATUS = isDone(H) returns a logical value, STATUS , when the
`MultimediaFileReader` object, H , reaches the end of the multimedia
file. If you set the `PlayCount` property to a value greater than 1 , STATUS
is `true` each time the object reaches the end of the file. STATUS is the
same as the EOF output value in the process method syntax.

Purpose   Locked status for input attributes and nontunable properties

> **Note** MultimediaFileReader will be removed in a future release. Use
> AudioFileReader instead. See dsp.AudioFileReader.isLocked.

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the MultimediaFileReader
object.

The isLocked method returns a logical value that indicates whether
input attributes and nontunable properties for the object are locked. The
object performs an internal initialization the first time the step method
is executed. This initialization locks nontunable properties and input
specifications, such as dimensions, complexity, and data type of the
input data. After locking, the isLocked method returns a true value.

signalblks.MultimediaFileReader.reset

Purpose   Reset internal states of multimedia file reader to read from beginning of file

---

Note MultimediaFileReader will be removed in a future release. Use AudioFileReader instead. See dsp.AudioFileReader.reset.

Syntax   reset(H)

Description reset(H) resets the MultimediaFileReader object to read from the beginning of the file.

Purpose   Output one frame of video frames, audio samples, or both to multimedia signal

---

Note `MultimediaFileReader` will be removed in a future release. Use `AudioFileReader` instead. See `dsp.AudioFileReader.step`.

Syntax   AUDIO = step(H)
I = step(H)
[I,AUDIO] = step(H)
[...,EOF] = step(H)
[Y,CB,CR] = step(H)
[Y,CB,CR,AUDIO] = step(H)

Description AUDIO = step(H) outputs one frame of audio samples, AUDIO. This behavior requires the AudioOutputPort property to be true and an input file that contains audio data. After the object plays the file the number of times PlayCount specifies, AUDIO contains silence.

I = step(H) outputs one frame of multidimensional video signal, I. To obtain this behavior, set the VideoOutputPort property to true and use an input file that contains video data.

[I,AUDIO] = step(H) outputs one frame of multidimensional video signal, I , and one frame of audio samples, AUDIO . This behavior requires the AudioOutputPort and VideoOutputPort properties to be true and an input file that contains audio and video data.

[...,EOF] = step(H) gives the end-of-file indicator in EOF. EOF is true each time the output contains the last audio sample and/or video frame in the file.

[Y,CB,CR] = step(H) outputs one frame of YCbCr 4:2:2 video data in the color components Y, CB, and CR. This behavior applies when you set the VideoOutputPort property to true, the ImageColorSpace property to YCbCr 4:2:2, and an input file which contains video data.

[Y,CB,CR,AUDIO] = step(H) outputs one frame of YCbCr 4:2:2 video data in the color components Y, CB, and CR, and one frame of audio samples in AUDIO. This applies when you set the AudioOutputPort and VideoOutputPort properties to true, the ImageColorSpace property to YCbCr 4:2:2, and an input file which contains audio and video data.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Write video frames, audio samples, or both to multimedia file

> **Note** `MultimediaFileWriter` will be removed in a future release. Use `AudioFileWriter` instead. See `dsp.AudioFileWriter`.

Description   The `MultimediaFileWriter` object writes video frames, audio samples, or both to a multimedia file.

To write video frames, audio samples, or both to a multimedia file:

1 Define and set up your multimedia file writer. See "Construction" on page 5-801.

2 Call `step` to write to a multimedia file according to the properties of `signalblks.MultimediaFileWriter`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = signalblks.MultimediaFileWriter` returns a multimedia file writer System object, `H`, that writes video frames, audio samples, or both to a multimedia file (such as an `AVI` file).

`H = signalblks.MultimediaFileWriter('`*PropertyName*`',`*PropertyValue*`,...)` returns a multimedia file writer System object, `H`, with each specified property set to the specified value.

`H = signalblks.MultimediaFileWriter(`*FILENAME*`,'`*PropertyName*`',`*PropertyValue*`` returns a multimedia file writer System object, `H`, with `Filename` property set to *FILENAME* and other specified properties set to the specified values.

Properties   **Filename**

Name of multimedia file to which to write

Specify the name of the multimedia file as a string. The default is `output.avi`.

FileFormat

Multimedia file format

Specify which multimedia file format the object writes. On Microsoft platforms, select `AVI`, `WAV`, `WMV`, or `WMA`. These abbreviations correspond to the following file formats:

- WAV: Microsoft WAVE Files

- WMV: Windows Media Video

- WMA: Windows Media Audio

- AVI: Audio-Video Interleave

The default setting for this property is AVI.

On all other platforms, the only selection is AVI.

AudioInputPort

Choose to write audio data

Use this property to specify whether the multimedia file writer object writes audio samples to a multimedia file. When you enable both this property and `VideoInputPort`, ensure that the video and audio input signals have the same frame period. Adjust the frame size (or number of rows) of the audio signal so that the frame period of the video signal is the same as the frame period of the audio signal. Calculate frame size by dividing the audio signal frequency by the frame rate of the video signal. The audio signal frequency is in samples per second (specified by `SampleRate`). The video signal frame rate is in frames per second (specified by `FrameRate`). The default selection for this property is `true`.

The multimedia file object takes a column vector as an input. Every column is a separate channel and each row corresponds to a single audio sample.

VideoInputPort

Choose to write video data

Use this property to specify whether the multimedia file writer object writes video frames to a multimedia file. When you enable both this property and the AudioInputPort property, ensure that the video and audio input signals have the same frame period. Adjust frame size (or number of rows) for the audio signal so the frame period is equal of both the video signal and the audio signal. Calculate frame size by dividing the audio signal frequency by the frame rate of the video signal. The audio signal frequency is in samples per second (specified by SampleRate). The video signal frame rate is in frames per second (specified by FrameRate). The default selection for this property is false

AudioCompressor

Algorithm that compresses audio data

Specify the type of compression algorithm the multimedia file writer uses to compress the audio data. Compression reduces the size of the multimedia file. Select None (uncompressed) to save uncompressed audio data to a multimedia file. The other options available reflect the audio compression algorithms installed on your system. This property is only available when writing WAV or AVI files on Windows platforms.

VideoCompressor

Algorithm that compresses video data

Specify the type of compression algorithm to multimedia file writer uses to compress the video data. This compression reduces the size of the multimedia file. Choose None (uncompressed) to save uncompressed video data to a multimedia file. The other options reflect the available video compression algorithms installed on your system. This property is only available when writing AVI files on Windows platforms.

SampleRate

Sampling rate of audio data stream

Specify the sampling rate of the input audio data as a positive numeric scalar. The default is 44100. This property only applies when you set the AudioInputPort property to true.

FrameRate

Frame rate of video data stream

Specify the frame rate of the video data in frames per second as a positive numeric scalar. The default is 30. This property only applies when you set the VideoInputPort property to true.

AudioDataType

Data type of the uncompressed audio

Specify the type of uncompressed audio data written to the file. This property only applies when writing uncompressed WAV files.

FileColorSpace

Color space to use when creating a file

Specify the color space of AVI files as RGB or YCbCr 4:2:2. This property only applies on Windows platforms when you set the FileFormat property to AVI. The default is RGB.

Methods	clone	Create multimedia file writer object with the same property values
	close	Release resources
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method

isLocked	Locked status for input attributes and nontunable properties
step	Write one frame of audio output samples

Examples   Decimate an audio signal, and write signal to disk as an AVI file:

```
hmfr = signalblks.MultimediaFileReader...
   ('AudioOutputDataType','double');
hfirdec = dsp.FIRDecimator; % decimate by 2
hmfw = signalblks.MultimediaFileWriter...
   ('speech_dft_11025.avi', ...
   'SampleRate', 22050/2);

while ~isDone(hmfr)
    audio = step(hmfr);
    audiod = step(hfirdec, audio);
    step(hmfw, audiod);
end

close(hmfr);
close(hmfw);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the To Multimedia File block reference page. The object properties correspond to the block parameters, except:

- The **Write** block parameter corresponds to both the AudioOutputPort and the VideoOutputPort object properties.

- The FrameRate and SampleRate properties are not available on the block. The block inherits these signals from the input line connected to the ports.

signalblks.MultimediaFileWriter

See Also   signalblks.MultimediaFileReader

5-806

Purpose   Create multimedia file writer object with the same property values

---

Note MultimediaFileWriter will be removed in a future release. Use AudioFileWriter instead. See dsp.AudioFileWriter.clone.

Syntax   C = clone(H)

Description   C = clone(H) creates a MultimediaFileWriter System object C, with the same property values as H. The clone method creates a new unlocked object.

signalblks.MultimediaFileWriter.close

Purpose   Release resources

> **Note** MultimediaFileWriter will be removed in a future release. Use
> AudioFileWriter instead. See dsp.AudioFileWriter.release.

Syntax   close(H)

Description close(H) releases system resources (such as memory, file handles or
hardware connections).

Purpose   Number of expected inputs to step method

> **Note** MultimediaFileWriter will be removed in a future release. Use
> AudioFileWriter instead. See dsp.AudioFileWriter.getNumInputs.

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, of
                   the step method

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

signalblks.MultimediaFileWriter.getNumOutputs

Purpose   Number of outputs of step method

> **Note** `MultimediaFileWriter` will be removed in a future release. Use
> `AudioFileWriter` instead. See `dsp.AudioFileWriter.getNumOutputs`.

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the `step`
method.

The `getNumOutputs` method returns a positive integer that is the
number of outputs from the `step` method. This value will change if you
alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

---

Note MultimediaFileWriter will be removed in a future release. Use AudioFileWriter instead. See dsp.AudioFileWriter.isLocked.

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the MultimediaFileWriter System object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

signalblks.MultimediaFileWriter.step

Purpose   Write one frame of audio output samples

> **Note** `MultimediaFileWriter` will be removed in a future release. Use
> `AudioFileWriter` instead. See `dsp.AudioFileWriter.step`.

Syntax
```
step(H,AUDIO)
step(H,I)
step(H,I,AUDIO)
step(H,Y,Cb,Cr)
step(H,Y,CB,CR)
step(H,Y,CB,CR,AUDIO)
```

Description   `step(H,AUDIO)` writes one frame of audio samples, `AUDIO` , to the output
file when you enable the `AudioInputPort` property. `AUDIO` is either a
vector or an M-by-2 matrix for mono or stereo inputs, respectively.

`step(H,I)` writes one frame of video, `I` , to the output file when you
enable the `VideoInputPort` property. `I` is an M-by-N-by-3 color video
signal.

`step(H,I,AUDIO)` writes one frame of video, `I` , and one frame of
audio samples, `AUDIO` , to the output file when you enable both the
`AudioInputPort` and `VideoInputPort` properties.

`step(H,Y,Cb,Cr)` writes one frame of video with Y, Cb, Cr components
to the file. This applies only when you set the `FileColorSpace` property
to `YCbCr 4:2:2`

`step(H,Y,CB,CR)` writes one frame of `YCbCr 4:2:2` data in the
color components Y, CB, and CR, to the output file when you set the
`VideoInputPort` property to `true`. The width of CB and CR must be
half of the width of Y, and the value of the `FileColorSpace` property
must be set to `YCbCr 4:2:2`.

`step(H,Y,CB,CR,AUDIO)` writes one frame of `YCbCr 4:2:2` data in
the color components Y, CB, and CR, and one frame of audio samples,
AUDIO, to the output file when you set both the `AudioInputPort` and

`VideoInputPort` properties to `true`. The width of `CB` and `CR` must be half of the width of `Y`, and the value of the `FileColorSpace` property must be set to `YCbCr 4:2:2`.

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

dsp.NCO

Purpose	Generate real or complex sinusoidal signals
Description	The numerically controlled oscillator, or NCO, object generates real or complex sinusoidal signals. The amplitude of the generated signal is always 1.

To generate real or complex sinusoidal signals:

1 Define and set up your NCO System object. See "Construction" on page 5-814.

2 Call step to generate the signals according to the properties of dsp.NCO. The behavior of step is specific to each object in the toolbox.

Construction	H = dsp.NCO returns an NCO System object, H, that generates a multichannel real or complex sinusoidal signal, with independent frequency and phase in each output channel.

H = dsp.NCO('*PropertyName*',*PropertyValue*,...) returns an NCO System object, H, with each specified property set to the specified value.

Properties **PhaseIncrementSource**

Source of phase increment

Indicate how to specify the phase increment: Property or Input port. The default is Input port.

PhaseIncrement

Phase increment

Specify the phase increment as an integer scalar. This property applies only when you set the PhaseIncrementSource property to Property. The default is 100.

PhaseOffsetSource

Source of phase offset

Specify the phase offset as `Property` or `Input port`. The default is `Property`.

PhaseOffset

Phase offset

Specify the phase offset as an integer scalar. This property applies only when you set the `PhaseOffsetSource` property to `Property`. The default is `0`.

Dither

Enable adding internal dithering to NCO algorithm

Set this property to `true` to add internal dithering to the NCO algorithm. Dithering is added using the PN Sequence Generator from the Communications System Toolbox product. The default is `true`.

NumDitherBits

Number of dither bits

Specify the number of dither bits as a positive integer. This property applies only when you set the `Dither` property to `true`. The default is `4`.

PhaseQuantization

Enable quantization of accumulated phase

Set this property to `true` to enable quantization of the accumulated phase. The default is `true`.

NumQuantizerAccumulatorBits

Number of quantizer accumulator bits

Specify the number of quantizer accumulator bits as an integer scalar greater than 1 and less than the accumulator word length. This property determines the number of entries in the lookup table of sine values. This property applies only when you set the `PhaseQuantization` property to `true`. The default is `12`.

PhaseQuantizationErrorOutputPort

Enable output of phase quantization error

Set this property to `true` to output the phase quantization error. This property applies only when you set the `PhaseQuantization` property to `true`. The default is `false`.

Waveform

Type of output signal

Specify the type of the output signal: `Sine`, `Cosine`, `Complex exponential` or `Sine and cosine`. The default is `Sine`.

SamplesPerFrame

Number of output samples per frame

Specify the number of samples per frame of the output signal. This property applies only when you set the `PhaseOffsetSource` property to `Property`. The default is 1. When the `PhaseOffsetSource` property is `Input port`, and the `PhaseIncrementSource` property is `Property`, the number of rows or frame size of the phase offset input determines the number of samples per frame of the output signal. When you set both the `PhaseOffsetSource` and `PhaseIncrementSource` properties to `Input port`, the number of rows in the inputs must be 1, and the samples per frame of the output signal is 1.

OutputDataType

Output data type

Specify the output data type as one of `double`, `single` or `Custom`. The default is `Custom`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

This constant property has a value `Floor`.

OverflowAction

Overflow action for fixed-point operations

This constant property has a value `Wrap`.

AccumulatorDataType

Accumulator word and fraction lengths

This constant property has a value `Custom`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as an unscaled `numerictype` object with a `Signedness` of `Auto`. The default is `numerictype([],16)`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,14)`.

Methods

clone	Create NCO object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
info	Characteristic information about generated signal

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset accumulator of NCO object
step	Sinusoidal signal with specified phase increment

Examples   Design an NCO source according to given specifications:

```
F0 = 510; % Output frequency = 510 Hz
df = 0.05; % Frequency resolution = 0.05 Hz
minSFDR = 96; % Spurious free dynamic range >= 96 dB
Ts = 1/8000; % Sample period = 1/8000 sec
dphi = pi/2; % Desired phase offset = pi/2;
```

Calculate number of accumulator bits required for the given frequency resolution

```
Nacc = ceil(log2(1/(df*Ts)));
% Actual frequency resolution achieved
actdf = 1/(Ts*2^Nacc);
% Calculate number of quantized accumulator bits
% required from the SFDR requirement
Nqacc = ceil((minSFDR-12)/6);
% Calculate the phase increment
phIncr = round(F0*Ts*2^Nacc);
% Calculate the phase offset
phOffset = 2^Nacc*dphi/(2*pi);

hnco = dsp.NCO('PhaseIncrementSource', 'Property', ...
'PhaseIncrement', phIncr,...
'PhaseOffset', phOffset,...
'NumDitherBits', 4, ...
'NumQuantizerAccumulatorBits', Nqacc,...
```

```
'SamplesPerFrame', 1/Ts, ...
'CustomAccumulatorDataType', numerictype([],Nacc));

y = step(hnco);
% Plot the mean-square spectrum of the 510 Hz sinewave
% generated by the NCO
sperd = spectrum.periodogram('hann'); ...
sperd.SamplingFlag = 'periodic';
msspectrum(sperd,double(y),'Fs',1/Ts);
```

Algorithms This object implements the algorithm, inputs, and outputs described on the NCO block reference page. The object properties correspond to the block properties, except:

There is no object property that corresponds to the **Sample time** block parameter. The objects assumes a sample time of one second.

See Also dsp.SineWave

dsp.NCO.clone

Purpose   Create NCO object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates an NCO system object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs to the
                     step method.

                     The getNumInputs method returns a positive integer that is the number
                     of expected inputs (not counting the object itself) to the step method.
                     This value will change if you alter any properties that turn inputs on or
                     off. You must call the step method with the number of input arguments
                     equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   `N = getNumOutputs(H)`

Description   `N = getNumOutputs(H)` returns the number of outputs, `N`, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Characteristic information about generated signal

Syntax   S = info(H)

Description S = info(H) returns a structure containing characteristic information, S, about the signal being generated by the NCO System object, H. The number of fields of S, and their values vary depending on the property value settings of H. For description of the possible fields and their values, see the following table.

Field	Value
NumPointsLUT	Number of data points for lookup table. The lookup table is implemented as a quarter-wave sine table.
SineLUTSize	Quarter-wave sine lookup table size in bytes.
TheoreticalSFDR	Theoretical spurious free dynamic range (SFDR) in dBc. This field applies when you set the PhaseQuantization property to true.
FrequencyResolution	Frequency resolution of the NCO in Hz. The sample time of the output signal is assumed to be 1 sec.

dsp.NCO.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the NCO System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles,
                   and hardware connections, and lets you change any properties or input
                   characteristics.

> **Note** You can use the release method on a System object in code
> generated from MATLAB, but once you release its resources, you cannot
> use that System object again.

Purpose   Reset accumulator of NCO object

Syntax   reset(H)

Description reset(H) resets the accumulator of the NCO System object, H, to zero.

Purpose   Sinusoidal signal with specified phase increment

Syntax   Y = step(H,INC)
               Y = step(H)
               Y = step(H,OFFSET)
               Y = step(H,INC,OFFSET)
               [Y, QERR] = step(H,...)

Description Y = step(H,INC) returns a sinusoidal signal, Y , generated by the NCO
               with the specified phase increment, INC. INC must be a built-in integer
               or a fi object comprising either a scalar or a row vector, where each row
               element corresponds to a separate channel.

               Y = step(H) when the PhaseIncrementSource and
               PhaseOffsetSource properties are both Property, returns a
               sinusoidal signal, Y.

               Y = step(H,OFFSET) returns a sinusoidal signal, Y, with phase offset,
               OFFSET, when the PhaseOffsetSource property is Input port. OFFSET
               must be a built-in integer or a fi object. The number of rows of OFFSET
               determines the number of samples per frame of the output signal. The
               number of columns of the OFFSET determines the number of channels
               of the output signal.

               Y = step(H,INC,OFFSET) returns a sinusoidal signal, Y, with
               phase increment, INC, and phase offset, OFFSET, when the
               PhaseIncrementSource and the PhaseOffsetSource properties are
               both Input port. INC and OFFSET must both be row vectors of the
               same length, where the length determines the number of channels in
               the output signal.

               [Y, QERR] = step(H,...) when the PhaseQuantization and the
               PhaseQuantizationErrorOutputPort properties are both true, returns
               a sinusoidal signal, Y, and output quantization error, QERR.

> **Note** The object performs an initialization the first time the step
> method is executed. This initialization locks nontunable properties and
> input specifications, such as dimensions, complexity, and data type
> of the input data. If you change a nontunable property or an input
> specification, the System object issues an error. To change nontunable
> properties or inputs, you must first call the release method to unlock
> the object.

Purpose   Vector normalization along specified dimension

Description The Normalizer object performs vector normalization along rows, columns, or specified dimension.

To perform vector normalization:

1 Define and set up your normalization object. See "Construction" on page 5-829.

2 Call step to perform vector normalization according to the properties of dsp.Normalizer. The behavior of step is specific to each object in the toolbox.

Construction H = dsp.Normalizer returns a normalization object, H, that normalizes the input over each column by the squared 2-norm of the column plus a bias term of 1e-10 used to protect against divide-by-zero.

H = dsp.Normalizer('*PropertyName*',*PropertyValue*, ...) returns a normalization object, H, with each property set to the specified value.

Properties **Method**

Type of normalization to perform

Specify the type of normalization to perform as 2-norm or Squared 2-norm. The 2-norm mode supports floating-point signals only. The Squared 2-norm supports both fixed-point and floating-point signals. The default is Squared 2-norm.

Bias

Real number added in denominator to avoid division by zero

Specify the real number to add in the denominator to avoid division by zero. The default is 1e-10. This property is tunable.

Dimension

Dimension to operate along

Specify whether to normalize along `Column` , `Row`, or `Custom`. The default is `Column`.

CustomDimension

Numerical dimension to operate along

Specify the one-based value of the dimension over which to normalize. The value of this parameter cannot exceed the number of dimensions in the input signal. This property applies when `Dimension` property is `Custom`. The default is 1.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling`, `Convergent`, `Floor` , `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. The default is `Wrap`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of `Same as input` or `Custom`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default to `numerictype([],32,32)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of `Same as product`, `Same as input`, `Custom`. The default is `Same as product`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of `Same as accumulator`, `Same as product`, `Same as input`, `Custom`. The default is `Same as product`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],32,32)`.

Methods

clone	Create normalization object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method

dsp.Normalizer

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Normalize input along specified dimension

Examples   Normalize a matrix:

```
hnorm = dsp.Normalizer;
x = magic(3);
y = step(hnorm, x);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Normalization block reference page. The object properties correspond to the block parameters, except:

- **Treat sample-based row input as column** — The block allows you to input a row vector and normalize the row vector as a column vector. The normalization object always normalizes along the value of the Dimension property.

- The normalization object does not support the **Minimum** and **Maximum** options for data output.

See Also   dsp.ArrayVectorMultiplier

Purpose   Create normalization object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a normalizer object, C, with the same property
                   values. The clone method creates a new unlocked object.

dsp.Normalizer.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   `N = getNumInputs(H)` returns the number of expected inputs to the `step` method.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                    method.

                    The getNumOutputs method returns a positive integer that is the
                    number of outputs from the step method. This value will change if you
                    alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   `isLocked(H)`

Description   `isLocked(H)` returns the locked state of the normalization object.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose	Normalize input along specified dimension
Syntax	Y = step(H,X)
Description	Y = step(H,X) returns a normalized outputY . The input X must be floating-point for the 2-norm mode, and either fixed-point and floating-point for the Squared 2-norm mode.

> **Note** The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Determine extrema (maxima or minima) in input signal

Description The PeakFinder object determines the extrema (maxima or minima) in the input signal.

To compute the extrema in the input signal:

1 Define and set up your peak finder. See "Construction" on page 5-839.

2 Call step to compute the extrema according to the properties of dsp.PeakFinder. The behavior of step is specific to each object in the toolbox.

Construction H = dsp.PeakFinder returns a peak finder System object, H, that compares the current signal value to the previous and next values to determine if the current value is an extremum.

H = dsp.PeakFinder('*PropertyName*',*PropertyValue*,...) returns a peak finder System object, H, with each specified property set to the specified value.

Properties   **PeakType**

Looking for maxima, minima, or both

Specify whether the object is looking for maxima, minima, or both. You can set this property to oMaxima, Minima, Maxima and Minima. The default is Maxima and Minima.

PeakIndicesOutputPort

Enable output of extrema indices

Set this property to true to output the extrema indices. The default is false.

PeakValuesOutputPort

Enable output of extrema values

Set this property to true to output the extrema values. The default is false.

MaximumPeakCount

Number of extrema to look for in each input signal

The object stops searching the input signal for extrema after the maximum number of extrema has been found. The value of this property must be an integer greater than or equal to one. The default is 10.

IgnoreSmallPeaks

Enable ignoring peaks below threshold

Set this property to `true` if you want to eliminate the detection of peaks whose amplitudes are within a specified threshold of neighboring values. The default is `false`.

PeakThreshold

Threshold below which peaks are ignored

Specify the noise threshold value. This property defines the current input value as a maximum if:

- The current input value minus the previous input value is greater than the threshold, and

- The current input value minus next input value is greater than the threshold.

This property applies when you set the `IgnoreSmallPeaks` property to `true`. The default is 0.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

The rounding method is a constant property set to `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

Methods

clone	Create peak finder object with same property values
getNumInputs	Number of expected inputs to the step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Number of extrema in input signal

Examples

Determine whether each value of an input signal is local maximum or minimum:

```
hpeaks1 = dsp.PeakFinder;
hpeaks1.PeakIndicesOutputPort = true;
hpeaks1.PeakValuesOutputPort = true;

x1 = [9 6 10 3 4 5 0 12]';
% Find the peaks of each input
% [prev;cur;next]: {[9;6;10],[6;10;3],...}
[cnt1, idx1, val1, pol1] = step(hpeaks1, x1);
```

Determine peak values for a fixed-point input signal:

```
hpeaks2 = dsp.PeakFinder('PeakType', 'Maxima', ...
```

```
                     'PeakValuesOutputPort', true, ...
                     'MaximumPeakCount', 2, ...
                     'IgnoreSmallPeaks', true, ...
                     'PeakThreshold', 0.25, ...
                     'OverflowAction', 'Saturate');
                     x2 = fi([-1;0.5;0],true,16,15);
                     [cnt2, val2] = step(hpeaks2, x2);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Peak Finder block reference page. The object properties correspond to the block parameters.

See Also   dsp.Maximum | dsp.Minimum

Purpose   Create peak finder object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a PeakFinder System object C, with the same
property values as H. The clone method creates a new unlocked object
with uninitialized states.

dsp.PeakFinder.getNumInputs

Purpose   Number of expected inputs to the step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs to the
                   step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   `N = getNumOutputs(H)`

Description   `N = getNumOutputs(H)` returns the number of outputs, `N`, of the `step` method

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

dsp.PeakFinder.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the `PeakFinder` System object.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.PeakFinder.step

Purpose	Number of extrema in input signal
Syntax	YCNT = step(H,X) [YCNT,IDX] = step(H,X) [..., VAL] = step(H,X) [..., POL] = step(H,X)
Description	YCNT = step(H,X) returns the number of extrema (minima, maxima, or both) YCNT in input signal X.

[YCNT,IDX] = step(H,X) when the PeakIndicesOutputPort property is true, returns the number of extrema YCNT and peak indices IDX in input signal X.

[..., VAL] = step(H,X) also returns the peak values VAL in input signal X when the PeakValuesOutputPort property is true.

[..., POL] = step(H,X) also returns the extrema polarity POL in input signal X when the PeakType property is Maxima and Minima and the PeakIndicesOutputPort property is true. The polarity is 1 for maxima and 0 for minima.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Peak-to-peak value

Description   The `PeakToPeak` object computes the peak-to-peak value of an input.

To obtain the peak-to-peak value:

1 Define and set up your peak-to-peak calculation. See "Construction" on page 5-849.

2 Call `step` to compute the peak-to-peak value for an input vector according to the properties of `dsp.PeakToPeak`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.PeakToPeak` creates a peak-to-peak System object, `H`, that computes the difference between the maximum and minimum value in an input or a sequence of inputs.

`H = dsp.PeakToPeak('`*PropertyName*`',`*PropertyValue*`,...)` returns an `PeakToPeak` System object, `H`, with each specified property set to the specified value.

Properties   **CustomDimension**

Dimension to operate along. Specify the dimension as a positive integer along which the peak-to-peak difference is computed. The value of this property cannot exceed the number of dimensions in the input signal. This property applies when the `Dimension` property is `'Custom'`.

Default: `1`

Dimension

Dimension to operate along. Specify the dimension along which to calculate the peak-to-peak ratio as one of `'All'`, `'Row'`, `'Column'`, or `'Custom'`. This property applies when the `RunningPeakToPeak` property is `false`. If you set this property to `'Custom'`, specify the dimension using the `CustomDimension` property.

Default: `'Column'`

FrameBasedProcessing

Process input in frames or as samples. Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. This property applies when the `RunningPeakToPeak` property is `true`.

Default: `true`

ResetCondition

Reset condition for running peak-to-peak mode. Specify the event to reset the running peak-to-peak as one of `'Rising edge'`, `'Falling edge'`, `'Either edge'`, or `'Non-zero'`. This property applies when the `ResetInputPort` property is `true`.

ResetInputPort

Enables resetting in running peak-to-peak mode. Set this property to `true` to enable resetting the running peak-to-peak. When the property is set to `true`, a reset input must be specified to the call of `step` to reset the running peak-to-peak difference. This property applies when the `RunningPeakToPeak` property is `true`.

Default: `false`

RunningPeakToPeak

Calculation over successive calls to `step`. Set this property to `true` to enable the calculation of the peak-to-peak difference over successive calls to the `step`.

Default: `false`

Methods

clone	Clones the current instance of the peak-to-peak object
getNumInputs	Number of expected inputs to the method
getNumOutputs	Number of outputs of the method
isLocked	Locked status (logical) for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset the running peak-to-peak object
step	Compute peak-to-peak value

Examples

Peak-to-Peak Value of Vector

Determine the peak-to-peak value for a vector input.

```
in1 = (1:10)';
h1 = dsp.PeakToPeak;
y1 = step(h1, in1);
```

Peak-to-Peak Value of Matrix Input

Determine the peak-to-peak value of a matrix input.

```
in2 = magic(4);
h2 = dsp.PeakToPeak;
h2.Dimension = 'All';
y2 = step(h2, in2)
```

References

[1] *IEEE® Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

dsp.PeakToPeak

See Also   dsp.Maximum **|** dsp.PeakToRMS **|**

Purpose   Clones the current instance of the peak-to-peak object

Syntax   clone(H)

Description   clone(H) clones the current instance of the peak-to-peak object, H.

dsp.PeakToPeak.getNumInputs

Purpose   Number of expected inputs to the `step` method

Syntax   `N = getNumInputs(H)`

Description   `N = getNumInputs(H)` returns the number of expected inputs, `N`, to the `step` method for the peak-to-peak object `H`.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of the step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method for the peak-to-peak object, H.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

dsp.PeakToPeak.isLocked

Purpose   Locked status (logical) for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the peak-to-peak object H.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.PeakToPeak.reset

Purpose   Reset the running peak-to-peak object

Syntax   reset(H)

Description   reset(H) resets the running peak-to-peak value for the object H.

Purpose   Compute peak-to-peak value

Syntax   Y = step(H,X)
Y = step(H,X,R)

Description   Y = step(H,X) computes the peak-to-peak value, Y, of the floating-point input vector X. When the RunningPeakToPeak property is true, Y corresponds to the peak-to-peak value of the input elements over successive calls to the step method.

Y = step(H,X,R) computes the peak-to-peak value of the input elements over successive calls to the step method. The object optionally resets its state based on the reset input signal, R, and the value of the ResetCondition property. This is possible when you set both the RunningPeakToPeak and the ResetInputPort properties to true.

dsp.PeakToRMS

Purpose	Peak-to-root-mean-square value of vector

Description   The `PeakToRMS` object calculates the peak-to-root-mean-square ratio of a vector.

To compute the peak-to-root-mean-square ratio:

1 Define and set up your peak-to-root-mean-square calculation. See "Construction" on page 5-860.

2 Call `step` to calculate the peak-to-root-mean-square value for an input vector according to the properties of `dsp.PeakToRMS`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.PeakToRMS` creates a peak-to-root-mean-square System object, `H`, that returns the ratio of the maximum magnitude (peak) to the root-mean-square (RMS) value in an input or a sequence of inputs.

`H = dsp.PeakToRMS('`*PropertyName*`',`*PropertyValue*`,...)` returns an `PeakToRMS` System object, `H`, with each specified property set to the specified value.

Properties   **CustomDimension**

Numerical dimension to operate along. Specify the dimension as a positive integer along which the peak-to-RMS ratio is computed. The value of this property cannot exceed the number of dimensions in the input signal. This property applies when `Dimension` property is `'Custom'`.

Default: 1

DecibelScaledOutput

Report output in decibels (dB). Set this property to `true` to enable output in dB. Set this property to `false` to report output as a ratio.

Default: `true`

Dimension

Dimension to operate along. Specify the dimension along which to calculate the peak-to-RMS ratio as one of `'All'`, `'Row'`, `'Column'`, or `'Custom'`. This property applies when the `RunningPeakToRMS` property is `false`. If you set this property to `'Custom'`, specify the dimension using the `CustomDimension` property.

Default: `'Column'`

FrameBasedProcessing

Process input in frames or as samples. Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. This property applies when the `RunningPeakToRMS` property is `true`.

Default: `true`

ResetCondition

Reset condition for running peak-to-RMS mode. Specify the event to reset the running peak-to-RMS as one of `'Rising edge'`, `'Falling edge'`, `'Either edge'`, or `'Non-zero'`. This property applies when the `ResetInputPort` property is `true`.

ResetInputPort

Enables resetting in running peak-to-RMS mode. Set this property to `true` to enable resetting. When the property is set to `true`, a reset input must be specified in the call to `step` to reset the running peak-to-RMS ratio. This property applies when the `RunningPeakToRMS` property is `true`.

Default: `false`

RunningPeakToRMS

Calculation over successive calls to step. Set this property to
true to enable the calculation of the peak-to-RMS ratio over
successive calls to the step.

Default: false

Methods	clone	Clones the current instance of the peak-to-root-mean-square object
	getNumInputs	Number of expected inputs to the method
	getNumOutputs	Number of outputs of the method
	isLocked	Locked status (logical) for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	reset	Reset the running peak-to-root-mean-square object
	step	Compute peak-to-RMS ratio

Definitions   **Peak-magnitude-to-RMS Level**

The peak-magnitude-to-RMS level is

$$\frac{||X||_\infty}{\sqrt{\frac{1}{N}\sum_{n=1}^{N}|X_n|^2}}$$

where the *l*-infinity norm and RMS values are computed along the
specified dimension.

Examples   **Peak-to-RMS Ratio of Vector Input**

Determine the peak-to-RMS ratio of a vector input.

```
in1 = (1:10)';
h1 = dsp.PeakToRMS;
y1 = step(h1,in1);
```

Peak-to-RMS Ratio of Matrix Input

Determine the peak-to-RMS ratio of a matrix input.

```
in2 = magic(4);
h2 = dsp.PeakToRMS;
h2.Dimension = 'All';
y2 = step(h2, in2)
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms,*
                   IEEE Standard 181, 2003.

See Also   dsp.Maximum **|** dsp.PeakToPeak **|**

dsp.PeakToRMS.clone

Purpose   Clones the current instance of the peak-to-root-mean-square object

Syntax   `clone(H)`

Description   `clone(H)` clones the current instance of the peak-to-root-mean-square (RMS) object `H`.

Purpose   Number of expected inputs to the step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method for the peak-to-root-mean-square object H.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

dsp.PeakToRMS.getNumOutputs

Purpose   Number of outputs of the `step` method

Syntax   `N = getNumOutputs(H)`

Description   `N = getNumOutputs(H)` returns the number of outputs, N, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status (logical) for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the peak-to-root-mean-square object H.

dsp.PeakToRMS.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset the running peak-to-root-mean-square object

Syntax   `reset(H)`

Description   `reset(H)` resets the running peak-to-root-mean-square value for the object `H`.

Purpose   Compute peak-to-RMS ratio

Syntax   `Y = step(H,X)`
`Y = step(H,X,R)`

Description   `Y = step(H,X)` computes the peak-to-RMS ratio, `Y`, of the floating-point input vector `X`. When the `RunningPeakToRMS` property is `true`, `Y` corresponds to the peak-to-RMS ratio of the input elements over successive calls to the `step` method.

`Y = step(H,X,R)` computes the peak-to-RMS ratio of the input elements over successive calls to the `step` method. The object optionally resets its state based on the reset input signal, `R`, and the value of the `ResetCondition` property. This is possible when you set both the `RunningPeakToRMS` and the `ResetInputPort` properties to `true`.

Purpose   Unwrap signal phase

Description   The PhaseUnwrapper object unwraps the signal phase input specified in radians.

To unwrap the signal phase input:

1 Define and set up your System object. See "Construction" on page 5-871.

2 Call step to unwrap the signal phase according to the properties of dsp.PhaseUnwrapper. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.PhaseUnwrapper returns a System object, H. This System object adds or subtracts appropriate multiples of 2π to each input element to remove phase discontinuities (unwrap).

H = dsp.PhaseUnwrapper('*PropertyName*',*PropertyValue*,...) returns an unwrapped System object, H, with each specified property set to the specified value.

Properties   **InterFrameUnwrap**

Enable unwrapping of phase discontinuities between successive frames

Set this property to false to unwrap phase discontinuities only within the frame. Set this property to true to also unwrap phase discontinuities between successive frames. This property applies when you set the FrameBasedProcessing to true. The default is true.

Tolerance

Jump size as true phase discontinuity

Specify the jump size that the phase unwrapper recognizes as a true phase discontinuity. The default is set to π (rather than a smaller value) to avoid altering legitimate signal features.

To increase the phase wrapper sensitivity, set the Tolerance property to a value slightly less than π.

FrameBasedProcessing

Enable frame-based processing

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. The default is `true`.

Methods

clone	Create phase unwrapper object with same property values
getNumInputs	Number of expected inputs to the method
getNumOutputs	Number of outputs of the method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of Phase Unwrapper System object
step	Unwrap input phase signal input

Examples

Unwrap input phase data:

```
hunwrap = dsp.PhaseUnwrapper;
p = [0 2/5 4/5 -4/5 -2/5 0 2/5 4/5 -4/5 -2/5 0 2/5 ...
 4/5 -4/5, -2/5]*pi;
y = step(hunwrap, p');
figure,stem(p); hold
stem(y, 'r');
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Unwrap block reference page. The object properties correspond to the Simulink block parameters.

Objects and blocks interpret frames differently. Objects process inputs as frames or as samples by setting the FrameBasedProcessing property. Blocks process inputs as frames or as samples by inheriting the frame information from the input ports. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   unwrap

Purpose   Create phase unwrapper object with same property values

Syntax   clone(H)

Description   clone(H) clones the current instance of the phase unwrapper object H.

Purpose   Number of expected inputs to the step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs to the
step method

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.PhaseUnwrapper.getNumOutputs

Purpose   Number of outputs of the `step` method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of the `step` method

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the phase unwrapper object H.

dsp.PhaseUnwrapper.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset internal states of Phase Unwrapper System object

Syntax   reset(H)

Description   reset(H) resets the 2π multiplier k to 0 for the phase unwrapper
                   object H. See "Frame-Based Processing" on page 2-1630 for details.

Purpose   Unwrap input phase signal input

Syntax   `Y = step(H,X)`

Description   `Y = step(H,X)` unwraps the input signal `X` in radians for the root mean square (RMS) object `H`.

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose   Pulse metrics of bilevel waveforms

Description   The `PulseMetrics` object computes rise times, fall times, pulse widths, and cycle metrics including pulse period, pulse separation, and duty cycle for bilevel waveforms.

To obtain pulse metrics for a bilevel waveform:

1 Define and set up your pulse metrics. See "Construction" on page 5-881.

2 Call `step` to compute the pulse metrics according to the properties of `dsp.PulseMetrics`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.PulseMetrics` creates a pulse metrics System object, `H`. The object computes the rise time, fall time, and width of a pulse. `dsp.PulseMetrics` also computes cycle metrics such as pulse separations, periods, and duty cycles. Because a pulse contains two transitions, the object contains a superset of the capability defined in `dsp.TransitionMetrics`.

`H = dsp.PulseMetrics('`*PropertyName*`',`*PropertyValue*`,...)` returns a `PulseMetrics` System object, `H`, with each specified property set to the specified value.

Properties   **CycleOutputPort**

        Enable cycle metrics. If `CycleOutputPort` is `true`, cycle metrics are reported for each pulse period.

        Default: `false`

    MaximumRecordLength

        Maximum samples to preserve between calls to `step`. This property requires a positive integer that specifies the maximum number of samples to save between calls to the step method. When the number of samples to be saved exceeds this length, the

oldest excess samples are discarded. This property applies when the RunningMetrics property is true and is tunable.

Default: 1000

PercentReferenceLevels

Lower-, middle-, and upper-percent reference levels. This property contains a three-element numeric row vector that contains the lower-, middle-, and upper-percent reference levels. These reference levels are used as an offset between the lower and upper states of the waveform when computing the duration of each transition.

Default: [10 50 90]

PercentStateLevelTolerance

Tolerance of the state level (in percent). This property requires a scalar that specifies the maximum deviation from either the low or high state before it is considered to be outside that state. The tolerance is expressed as a percentage of the waveform amplitude.

Default: 2

Polarity

Polarity of pulse to extract. This property specifies the type of pulse to extract by the polarity of the leading transition. Valid values for this property are 'positive' or 'negative'.

Default: 'positive'

PostshootOutputPort

Enable posttransition aberration metrics. If this property is set to true, overshoot and undershoot metrics are reported for a region defined immediately after each transition. The posttransition aberration region is defined as the waveform interval that begins

at the end of each transition and whose duration is the value of
PostshootSeekFactor times the computed transition duration. If
a complete subsequent transition is detected before the interval
is over, the region is truncated at the start of the subsequent
transition. The metrics are computed for each transition that has
a complete posttransition aberration region.

Default: false

PostshootSeekFactor

Corresponds to the duration of time to search for the overshoot and
undershoot metrics immediately following each transition. The
duration is expressed as a factor of the duration of the transition.
This property is enabled only when the PostshootOutputPort
property is set to true and is tunable.

Default: 3

PreshootOutputPort

Enable pretransition aberration metrics. If PreshootOutputPort
is set to true, overshoot and undershoot metrics are reported
for a region defined immediately before each transition. The
pretransition aberration region is defined as the waveform
interval that ends at the start of each transition and whose
duration is PreshootSeekFactor times the computed transition
duration.

Default: 'false'

PreshootSeekFactor

Corresponds to the duration of time to search for the overshoot and
undershoot metrics immediately preceding each transition. The
duration is expressed as a factor of the duration of the transition.
This property is enabled only when the PreshootOutputPort
property is set to true and is tunable.

Default: 3

RunningMetrics

Enable metrics over all calls to `step`. If `RunningMetrics` is set to `false`, metrics are computed for each call to `step` independently. If `RunningMetrics` is set to `true`, metrics are computed across subsequent calls to `step`. If there are not enough samples to compute metrics associated with the last transition, posttransition aberration region, or settling seek duration in the current record, the object will defer reporting all transition, aberration, and settling metrics associated with the last transition until a subsequent call to `step` is made with enough data to compute all enabled metrics for that transition.

Default: `false`

SampleRate

Sampling rate of uniformly-sampled signal. Specify the sample rate in hertz as a positive scalar. This property is used to construct the internal time values that correspond to the input sample values. Time values start with zero. This property applies when the `TimeInputPort` property is set to `false`.

Default: 1

SettlingOutputPort

Enable settling metrics. If the `SettlingOutputPort` property is set to `true`, settling metrics are reported for each transition. The region used to compute the settling metrics starts at the midcrossing and lasts until the `SettlingSeekDuration` has elapsed. If an intervening transition occurs, or the signal has not settled within the `PercentStateLevelTolerance` of the final level, `NaN` is returned for each metric. If there are not enough samples after the last transition to complete the `SettlingSeekDuration`, no metrics are reported for the last

transition. The metrics are reported for the transition the next time `step` is called if the `RunningMetrics` property is set to `true`.

Default: `false`

SettlingSeekDuration

Duration of time over which to search for settling. This property value is a scalar that specifies the amount of time to inspect from the mid-reference level crossing (in seconds). If the transition has not yet settled, or a subsequent complete transition is detected within this duration, the `PulseMetrics` object will report `NaN` for all settling metrics. This property is tunable and applies only when you set the `SettlingOutputPort` property to `true`.

Default: `0.02`

StateLevels

Low- and high-state levels. This property is a 2–element numeric row vector that contains the low- and high-state levels respectively. These state levels correspond to the nominal logic low and high levels of the pulse waveform. This property is tunable.

Default: `[0 2.3]`

StateLevelsSource

Auto or manual state-level computation. If `StateLevelsSource` is set to `'Auto'`, the first record sent to `step` is sent to `dsp.StateLevels` with the default settings to determine the state levels of the incoming waveform. If this property is set to `'Property'`, the object uses the values the user specifies in the `StateLevels` property.

Default: `'Property'`

TimeInputPort

> Add input to specify sample instants. Set `TimeInputPort` to `true` to enable an additional real input column vector to `step` to specify the sample instants that correspond to the sample values. If this property is `false`, the sample instants are built internally. The sample instants start at zero and increment by the reciprocal of the `SampleRate` property for subsequent samples. The sample instants continue to increment if the `RunningMetrics` property is set to `true` and no intervening calls to the `reset` or `release` methods are encountered.
>
> **Default:** `false`

TransitionOutputPort

> Enable transition metrics. If the `TransitionOutputPort` property is set to `true`, transition metrics are reported for the initial and final transitions of each pulse.
>
> **Default:** `false`

Methods		
	clone	Clones the current instance of the pulse metrics object
	getNumInputs	Number of expected inputs to the method
	getNumOutputs	Number of outputs of the method
	isLocked	Locked status (logical) for input attributes and nontunable properties
	plot	Plots signal and metrics computed in last call to

reset	Reset the running pulse metrics object
step	Pulse metrics of bilevel waveforms

Examples

Width, Period, and Duty Cycle

Determine the width, period, and duty cycle of a 5 V pulse sampled at 4 MHz.

Load the data and sampling instants.

```
load('pulseex.mat', 'x', 't');
```

Construct your pulse metrics System object. Set the `TimeInputPort` property to `true` to specify the sampling instants as an input to `step`. Set the `CycleOutputPort` property to `true` to obtain metrics for each pulse. Because the input is a 5V pulse, set the `StateLevels` property to `[O 5]`.

```
hpm1 = dsp.PulseMetrics('TimeInputPort',true, ...
                        'CycleOutputPort', true, ...
                        'StateLevels',[O 5])
```

Call `step` to compute the cycle metrics and plot the result.

```
[pulse, cycle] = step(hpm1,x,t);
plot(hpm1)
text(t(2),-0.5,['Duty Cycle: ',num2str(cycle.DutyCycle)]);
```

Slew Rates for 2.3 V Digital Clock

Find the slew rates of the leading and trailing edges of a 2.3 V digital clock sampled at 4 MHz.

Construct your pulse metrics System object. Set the `TransitionOutputPort` property to `true` to report transition metrics for the initial and final transitions. Set the `StateLevelsSource` property to `'Auto'` to estimate the state levels from the data.

dsp.PulseMetrics

```
hpm2 = dsp.PulseMetrics('SampleRate',4e6, ...
                        'TransitionOutputPort', true, ...
                        'StateLevelsSource','Auto');
```

Compute the pulse and transition metrics and plot the result.

```
[pulse, transition] = step(hpm2,x);
plot(hpm2)
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

See Also   dsp.StateLevels **|** dsp.TransitionMetrics **|**

Purpose   Clones the current instance of the pulse metrics object

Syntax   clone(H)

Description   clone(H) clones the current instance of the pulse metrics object H.

dsp.PulseMetrics.getNumInputs

Purpose   Number of expected inputs to the `step` method

Syntax   `N = getNumInputs(H)`

Description   `N = getNumInputs(H)` returns the number of expected inputs, `N`, to the `step` method for the pulse metrics object `H`.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of the step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

dsp.PulseMetrics.isLocked

Purpose   Locked status (logical) for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the pulse metrics object H.

Purpose   Plots signal and metrics computed in last call to `step`

Syntax   `plot(H)`

Description   `plot(H)` plots the signal and metrics resulting from the last call of the step method.

By default `plot` displays

- the low and high state levels and the state-level boundaries defined by the `PercentStateLevelTolerance` property.

- the lower-, middle-, and upper-reference levels.

- the locations of the mid-reference level crossings of the positive (+) and negative (-) transitions of each detected pulse.

When the `TransitionOutputPort` property is set to `true`, the locations of the upper and lower crossings are also plotted. When the `PreshootOutputPort` or `PostShootOutputPort` properties are set to `true,` the corresponding overshoots and undershoots are plotted as inverted or noninverted triangles. When the `SettlingOutputPort` property is set to `true`, the locations where the signal enters and remains within the lower and upper state boundaries over the specified seek duration are plotted.

dsp.PulseMetrics.reset

Purpose   Reset the running pulse metrics object

Syntax   reset(H)

                   Description

                   reset(H) clears information from previous calls to step.

Purpose   Pulse metrics of bilevel waveforms

Syntax   PULSE = step(H,X)
                [PULSE,CYCLE] = step(H,X)
                [PULSE, TRANSITION] = step(H,X)
                [PULSE,PRESHOOT] = step(H,X)
                [PULSE,POSTSHOOT] = step(H,X)
                [PULSE,SETTLING] = step(H,X)
                [...] = step(H,X,T)

Description PULSE = step(H,X) returns a structure array, PULSE, whose fields
                contain real-valued column vectors. The number of rows of each field
                corresponds to the number of complete pulses found in the real-valued
                column vector input, X. Each pulse starts with a transition of the
                polarity specified by the Polarity property and ends with a transition
                of the opposite polarity.

                PULSE fields:

                • PositiveCross — Instants where the positive-going transitions cross
                  the mid-reference level of each pulse

                • NegativeCross — Instants where the negative-going transitions
                  cross the mid-reference level of each pulse

                • Width — Absolute difference between PositiveCross and
                  NegativeCross of each pulse

                • RiseTime — Duration between the linearly-interpolated instants
                  when the positive-going (rising) transition of each pulse crosses the
                  lower- and upper-reference levels

                • FallTime — Duration between the linearly-interpolated instants
                  when the negative-going (falling) transition of each pulse crosses the
                  upper- and lower-reference levels

                [PULSE,CYCLE] = step(H,X) returns a structure array, CYCLE,
                whose fields contain real-valued column vectors when you set the
                CycleOutputPort property to true. The number of rows of each field

corresponds to the number of complete pulse periods found in the real-valued column vector input, X. You need at least three consecutive alternating polarity transitions that start and end with the same polarity as the value of the Polarity property if you want to compute cycle metrics. If the last transition found in the input X does not match the polarity of the Polarity property, the pulse separation, period, frequency, and duty cycle are not reported for the last pulse. If the RunningMetrics property is set to true when this occurs, all pulse, cycle, transition, preshoot, postshoot, and settling metrics associated with the last pulse are deferred until a subsequent call to step detects the next transition.

CYCLE fields:

- Period — Duration between the first transition of the current pulse and the first transition of the next pulse

- Frequency — Reciprocal of the period

- Separation — Durations between the mid-reference level crossings of the second transition of each pulse and the first transition of the next pulse

- Width — Durations between the mid-reference level crossings of the first and second transitions of each pulse. This is equivalent to the width parameter of the PULSE structure.

- DutyCycle — Ratio of the width to the period for each pulse

[PULSE, TRANSITION] = step(H,X) returns a structure array, TRANSITION, when you set the TransitionOutputPort property to true. The fields of TRANSITION contain real-valued matrices with two columns which correspond to the metrics of the first and second transitions. The number of rows corresponds to the number of pulses found in the input waveform.

TRANSITION fields:

- Duration — Amount of time between the interpolated instants where the transition crosses the lower- and upper-reference levels

- `SlewRate` — Ratio of absolute difference between the upper and lower reference levels to the transition duration

- `MiddleCross` — Linearly-interpolated instant in time where the transition first crosses the mid-reference level

- `LowerCross` — Linearly-interpolated instant where the signal crosses the lower-reference level

- `UpperCross` — Linearly-interpolated instant where the signal crosses the upper-reference level

`[PULSE,PRESHOOT] = step(H,X)` returns a structure array, `PRESHOOT`, when you set the `PreshootOutputPort` property to `true`. The fields of `PRESHOOT` contain real-valued 2-column matrices whose row length corresponds to the number of transitions found in the input waveform. The field names are identical to those of the `POSTSHOOT` structure array.

`[PULSE,POSTSHOOT] = step(H,X)` returns a structure, `POSTSHOOT`, when you set the `PostshootOutputPort` property to `true`. The fields of `POSTSHOOT` contain real-valued 2-column matrices whose row length corresponds to the number of transitions found in the input waveform.

`PRESHOOT` and `POSTSHOOT` fields:

- `Overshoot` — Overshoot of the region of interest expressed as a percentage of the waveform amplitude

- `Undershoot` — Undershoot of the region of interest expressed as a percentage of the waveform amplitude

- `OvershootLevel` — Level of the overshoot

- `UndershootLevel` — Level of the undershoot

- `OvershootInstant` — Instant that corresponds to the overshoot

- `UndershootInstant` — Instant that corresponds to the undershoot

`[PULSE,SETTLING] = step(H,X)` returns a structure array, `SETTLING`, when you set the `SettlingOutputPort` property to `true`. The fields of `SETTLING` correspond to the settling metrics for each transition. Each

field is a column vector whose elements correspond to the individual settling durations, levels, and instants.

SETTLING fields:

- Duration — Amount of time from when the signal crosses the mid-reference level to the time where the signal enters and remains within the specified PercentStateLevelTolerance of the waveform amplitude over the specified settling seek duration

- Instant — Instant in time where the signal enters and remains within the specified tolerance

- Level — Level of the waveform where it enters and remains within the specified tolerance

The above operations can be used simultaneously, provided the System object properties are set appropriately. One example of providing all possible inputs and returning all possible outputs is :

[PULSE,CYCLE,TRANSITION,PRESHOOT,POSTSHOOT,SETTLING] = step(H,X) which returns the PULSE, CYCLE, TRANSITION, PRESHOOT, POSTSHOOT, and SETTLING structures when the CycleOutputPort, PreshootOutputPort, PostshootPort, and SettlingOutputPort properties are true. You may enable or disable any combination of output ports. However, the output arguments are defined in the order shown here.

[...] = step(H,X,T) performs the above metrics with respect to a sampled signal, whose sample values, X, and sample instants, T, are real-valued column vectors of the same length. The additional input T applies only when you set the TimeInputPort property to true.

Purpose   Convert reflection coefficients to autocorrelation coefficients

Description   The `RCToAutocorrelation` object converts reflection coefficients to autocorrelation coefficients.

To convert reflection coefficients to autocorrelation coefficients:

1 Define and set up your System object. See "Construction" on page 5-899.

2 Call `step` to convert reflection coefficients according to the properties of `dsp.RCToAutocorrelation`. The behavior of `step` is specific to each object in the toolbox.

Construction `H = dsp.RCToAutocorrelation` returns an `RCToAutocorrelation` System object, `H`. This object converts reflection coefficients to autocorrelation coefficients, assuming an error power of 1.

`H = dsp.RCToAutocorrelation('`*PropertyName*`',`*PropertyValue*`,...)` returns an object, `H`, that converts reflection coefficients into autocorrelation coefficients, with each specified property set to the specified value.

Properties   **PredictionErrorInputPort**

Enable prediction error power input

Choose how to select the prediction error power. When you set this property to `true`, you must specify the prediction error power as a second input to the `step` method. When you set this property to `false`, the object assumes a prediction error power of 1. The default is `false`.

dsp.RCToAutocorrelation

Methods

clone	Create RC to autocorrelation object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Convert columns of reflection coefficients to autocorrelation coefficients

Examples

Convert reflection coefficients to autocorrelation coefficients:

```
k = [-0.8091 0.2525 -0.5044 0.4295 -0.2804 0.0711].';
hrc2ac = dsp.RCToAutocorrelation;
ac = step(hrc2ac, k);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the LPC/RC to Autocorrelation block reference page. The object properties correspond to the block parameters, except:

- There is no object property that corresponds to the **Type of conversion** block parameter.

- **PredictionErrorInputPort** is a pop-up menu choice on the block. Setting the PredictionErrorInputPort object property to false corresponds to selecting Assume P = 1 in the pop-up menu. Setting PredictionErrorInputPort to true corresponds to selecting Via input port from the pop-up menu.

See Also   dsp.LPCToLSF | dsp.LPCToRC | dsp.LPCToCepstral |
                  dsp.LPCToAutocorrelation

dsp.RCToAutocorrelation.clone

Purpose   Create RC to autocorrelation object with same property values

Syntax   `C = clone(H)`

Description   `C = clone(H)` creates a `RCToAutocorrelation` System object `C`, with the same property values as `H`. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs to the
                   step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.RCToAutocorrelation.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the RCToAutocorrelation
System object.

The isLocked method returns a logical value that indicates whether
input attributes and nontunable properties for the object are locked. The
object performs an internal initialization the first time the step method
is executed. This initialization locks nontunable properties and input
specifications, such as dimensions, complexity, and data type of the
input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Convert columns of reflection coefficients to autocorrelation coefficients

Syntax   AC = step(H,K)
                   AC = step(H,K,P)

Description   AC = step(H,K) converts the columns of the reflection coefficients, K, to autocorrelation coefficients, AC.

AC = step(H,K,P) when you set the PredictionErrorInputPort property to true, converts the columns of the reflection coefficients, K , to autocorrelation coefficients, AC, using P as the prediction error power. P must be a row vector with same number of columns as in K.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

---

dsp.RCToLPC

Purpose
Convert reflection coefficients to linear prediction coefficients

Description
The RCToLPC object converts reflection coefficients to linear prediction coefficients.

To convert reflection coefficients to LPC:

1 Define and set up your RC to LPC System object. See "Construction" on page 5-908.

2 Call step to convert RC to LPC according to the properties of dsp.RCToLPC. The behavior of step is specific to each object in the toolbox.

Construction
H = dsp.RCToLPC returns an RC to LPC System object, H, that converts reflection coefficients (RC) to linear prediction coefficients (LPC).

H = dsp.RCToLPC('*PropertyName*',*PropertyValue*,...) returns an RC to LPC conversion object, H, with each specified property set to the specified value.

Properties

PredictionErrorOutputPort

Enable normalized prediction error power output

Set this property to true to return the normalized error power as a vector with one element per input channel. Each element varies between 0 and 1. The default is true.

ExceptionOutputPort

Produce output with stability status of filter represented by LPC coefficients

Set this property to true to return the stability of the filter. The output is a vector of a length equal to the number of channels. A logical value of 1 indicates a stable filter. A logical value of 0 indicates an unstable filter. The default is false.

Methods

clone	Creates instance of object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Convert columns of reflection coefficients to linear prediction coefficients

Examples

Convert reflection coefficients to linear prediction coefficients:

```
hlevinson = dsp.LevinsonSolver;
hac = dsp.Autocorrelator;
hac.MaximumLagSource = 'Property';
hac.MaximumLag = 10; % Compute autocorrelation
                     % lags between [0:10]
hrc2lpc = dsp.RCToLPC;

x = (1:100)';
a = step(hac, x);
k = step(hlevinson, a); % Compute reflection coefficients
[A, P] = step(hrc2lpc, k);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the LPC to/from RC block reference page. The object properties correspond to the block parameters, except:

There is no object property that corresponds to the **Type of conversion** block parameter. The object always converts LPC to RC.

.

dsp.RCToLPC

See Also   `dsp.LPCToRC` | `dsp.LPCToAutocorrelation`

Purpose   Creates instance of object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an RCToLPC System object C, with the same
                   property values as H. The clone method creates a new unlocked object.

dsp.RCToLPC.getNumInputs

Purpose　　　Number of expected inputs to step method

Syntax　　　`N = getNumInputs(H)`

Description　`N = getNumInputs(H)` returns the number of expected inputs to the `step` method

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the RCToLPC System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.RCToLPC.step

Purpose   Convert columns of reflection coefficients to linear prediction coefficients

Syntax
```
[A,P] = step(H,K)
A = step(H,K)
[..., S] = step(H,K)
```

Description   `[A,P] = step(H,K)` converts the columns of the reflection coefficients, `K`, to linear prediction coefficients, `A`, and outputs the normalized prediction error power, `P`.

`A = step(H,K)` when the `PredictionErrorOutputPort` property is `false`, converts the columns of the reflection coefficients, `K`, to linear prediction coefficients, `A`.

`[..., S] = step(H,K)` also outputs the LPC filter stability, `S`, when the `ExceptionOutputPort` property is `true`.

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose   Root mean square of vector elements

Description   The RMS object computes the root mean square (RMS) value.

To compute the RMS value of your input:

1 Define and set up your RMS calculation. See "Construction" on page 5-917.

2 Call step to compute the RMS value for an input according to the properties of dsp.RMS. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.RMS returns a System object, H, that computes the root mean square (RMS) of an input or a sequence of inputs over the specified Dimension.

H = dsp.RMS(*'PropertyName'*,*PropertyValue*,...) returns an RMS System object, H, with each specified property set to the specified value.

Properties   **RunningRMS**

Enable calculating RMS over time

Set this property to true to enable calculating the RMS over successive calls to the step method.

Default: false

ResetInputPort

Enable resetting in running RMS mode

Set this property to true to enable resetting the running RMS. When the property is set to true, you must specify a reset input to the step method to reset the running RMS. This property applies when you set the RunningRMS property to true.

Default: false

ResetCondition

Reset condition for running RMS mode

Specify the event to reset the running RMS as one of `Rising edge`, `Falling edge`, `Either edge`, or `Non-zero`. `Non-zero` resets the running RMS each time a nonzero sample is acquired. See "Rising and Falling Edges" on page 5-919 for definitions of rising and falling edges. This property applies when you set the `ResetInputPort` property to `true`.

Default: `Non-zero`

Dimension

Dimension to compute RMS value along

Specify the dimension along which to calculate the RMS as one of `All`, `Row`, `Column`, or `Custom`. This property applies only when you set the `RunningRMS` property to `false`. Specifying the `Dimension` property as `All` computes the RMS value over the entire input.

Default: `Column`

CustomDimension

Numerical dimension to operate along

Specify the dimension (one-based scalar integer value) of the input signal, along which the RMS is computed. The cannot exceed the number of dimensions in the input signal. This property applies when you set the `Dimension` property to `Custom`.

Default: 1

FrameBasedProcessing

Enable frame-based processing

Set this property to `true` to enable frame-based processing for 2-dimensional inputs. Set this property to `false` to

enable sample-based processing. The object always performs sample-based processing for N-D inputs where N is greater than 2. This property applies when you set the RunningRMS property to true.

Default: true

Methods

clone	Clones the current instance of the root mean square object
getNumInputs	Number of expected inputs to the method
getNumOutputs	Number of outputs of the method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset the running root mean square object
step	Root mean square of input

Definitions

Root-Mean-Square Level

The root-mean-square level of a vector, *X*, is

$$X_{\text{RMS}} = \sqrt{\frac{1}{N} \sum_{n=1}^{N} |X_n|^2}$$

with the summation performed along the specified dimension.

Rising and Falling Edges

A rising edge:

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

- Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero.

Rising edge

Rising edge

Rising edge

Rising edge

Not a rising edge because it continues a rise from a negative value to zero.

A falling edge:

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

- Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero.

Falling edge   Falling edge

Falling edge

Falling edge

Not a falling edge because it continues a fall from a positive value to zero.

Examples   **RMS Value of Vector Input**

Compute the RMS value of a vector consisting of the integers 1 to 10.

Create a row vector of the integers 1 to 10. Construct the RMS System object with the Dimension property set to `'literal'`. Compute the RMS value.

```
x = 1:10;
hrms = dsp.RMS('Dimension','row');
rmsval = step(hrms,x);
```

RMS Value of Matrix Input

Compute the RMS value of a matrix with the Dimension property set to `'All'`.

```
in2 = magic(4);
hrms2d = dsp.RMS;
hrms2d.Dimension = 'All';
y_rms2 = step(hrms2d, in2);
```

The output is equivalent to reshaping the 4-by-4 matrix into a 16-by-1, or 1-by-16 vector and computing the RMS value for the vector.

Algorithms   This object implements the algorithm, inputs, and outputs described on the RMS block reference page. The object properties correspond to the Simulink block parameters, except:

- **Treat sample-based row input as a column** block parameter is not supported by the dsp.RMS object.

- **Reset Port** block parameter corresponds to both the ResetCondition and the ResetInputPort object properties.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the FrameBasedProcessing property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

dsp.RMS

See Also   dsp.Mean | dsp.StandardDeviation | dsp.Variance.

Purpose   Clones the current instance of the root mean square object

Syntax   clone(H)

Description   clone(H) clones the current instance of the root mean square (RMS) object H

dsp.RMS.getNumInputs

Purpose   Number of expected inputs to the `step` method

Syntax   `N = getNumInputs(H)`

Description   `N = getNumInputs(H)` returns the number of expected inputs, `N`, to the `step` method for the root mean square (RMS) object `H`.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of the step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

dsp.RMS.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the root mean square (RMS) object H.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles,
                  and hardware connections, and lets you change any properties or input
                  characteristics.

Note You can use the release method on a System object in code
generated from MATLAB, but once you release its resources, you cannot
use that System object again.

---

dsp.RMS.reset

Purpose   Reset the running root mean square object

Syntax   `reset(H)`

Description   `reset(H)` resets the running root mean square (RMS) for the object `H`.

Purpose   Root mean square of input

Syntax   Y = step(H,X)
                Y = step(H,X,R)

Description Y = step(H,X) computes the root mean square (RMS) output, Y , of
                input vector X. When the RunningRMS property is true, Y corresponds to
                the RMS of the input elements over successive calls to the step method.

                Y = step(H,X,R) resets the running RMS state based on the value of R,
                the reset signal, and the ResetCondition property. This computation is
                possible when you set both the RunningRMS and the ResetInputPort
                properties to true.

                Note The object performs an initialization the first time the step
                method is executed. This initialization locks nontunable properties and
                input specifications, such as dimensions, complexity, and data type
                of the input data. If you change a nontunable property or an input
                specification, the System object issues an error. To change nontunable
                properties or inputs, you must first call the release method to unlock
                the object.

                ---

dsp.ScalarQuantizerDecoder

Purpose　　Convert each index value into quantized output value

Description　　The `ScalarQuantizerDecoder` object converts each index value into a quantized output value. The specified codebook defines the set of all possible quantized output values or codewords. Input index values less than 0 are set to 0 and index values greater $N-1$ are set to $N-1$. N is the length of the codebook vector.

To convert an index value into a quantized output value:

1 Define and set up your scalar quantizer decoder. See "Construction" on page 5-930.

2 Call `step` to convert the index value according to the properties of `dsp.ScalarQuantizerDecoder`. The behavior of `step` is specific to each object in the toolbox.

Construction　　`H = dsp.ScalarQuantizerDecoder` returns a scalar quantizer decoder System object, `H`, that transforms zero-based input index values into quantized output values.

`H = dsp.ScalarQuantizerDecoder('PropertyName',PropertyValue,...)` returns a scalar quantizer decoder object, `H`, with each specified property set to the specified value.

Properties　　**CodebookSource**

How to specify codebook values

Specify how to determine the codebook values as `Property` or `Input port`. The default is `Property`.

Codebook

Codebook

Specify the codebook as a vector of quantized output values that correspond to each index value. The default is `1:10`. This property is tunable.

OutputDataType

Data type of codebook and quantized output

Specify the data type of the codebook and quantized output values as `Same as input`, `double`, `single` or `Custom`. The default is `double`.

Fixed-Point Properties

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a signed or unsigned `numerictype` object. This property applies only when you set the `OutputDataType` property to `Custom`. The default is `numerictype(true,16)`.

Methods

clone	Create scalar quantizer decoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Operate on inputs to calculate outputs

Examples

Given a codebook and index values as inputs, determine the corresponding output quantized values:

```
codebook = single([-2.1655 -1.3238 -0.7365 -0.2249 0.2726, ...
```

```
                        0.7844 1.3610 2.1599]);
                        indices = uint8([1 3 5 7 6 4 2 0]);
                        hsqdec = dsp.ScalarQuantizerDecoder;
                        hsqdec.CodebookSource = 'Input port';
                        qout = step(hsqdec, indices, codebook);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Scalar Quantizer Decoder block reference page. The object properties correspond to the block parameters, except:

There is no object property that directly corresponds to the **Action for out of range index value** block parameter. The object sets any index values less than 0 to 0 and any index values greater than or equal to N to $N - 1$.

See Also   dsp.ScalarQuantizerEncoder | dsp.VectorQuantizerDecoder

Purpose   Create scalar quantizer decoder object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a `ScalarQuantizerDecoder` System object `C`, with the same property values as `H`. The clone method creates a new unlocked object with uninitialized states.

dsp.ScalarQuantizerDecoder.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                     method.

                     The getNumOutputs method returns a positive integer that is the
                     number of outputs from the step method. This value will change if you
                     alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   `isLocked(H)`

Description   `isLocked(H)` returns the locked state of the `ScalarQuantizerDecoder`
                   System object.

                   The `isLocked` method returns a logical value that indicates whether
                   input attributes and nontunable properties for the object are locked. The
                   object performs an internal initialization the first time the `step` method
                   is executed. This initialization locks nontunable properties and input
                   specifications, such as dimensions, complexity, and data type of the
                   input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.ScalarQuantizerDecoder.step

Purpose   Operate on inputs to calculate outputs

Syntax   Q = step(H, I)
                   Q = step(H,I,C)

Description   Q = step(H, I) returns the quantized output values Q corresponding
                   to the input indices I. The data type of I can be uint8, uint16, uint32,
                   int8, int16, or int32. The OutputDataType property determines the
                   data type for Q.

                   Q = step(H,I,C) uses input C as the codebook values when you set
                   the CodebookSource property to Input port. The data type of C can
                   be double, single, or fixed-point. The output Q has the same data
                   type as the codebook input C.

Purpose   Associate input value with index value of quantization region

Description   The `ScalarQuantizerEncoder` object encodes each input value by associating that value with the index value of the quantization region. Then, the object outputs the index of the associated region.

To encode an input value by associating it with an index value of the quantization region:

1 Define and set up your scalar quantizer encoder. See "Construction" on page 5-939.

2 Call `step` to encode the input value according to the properties of `dsp.ScalarQuantizerEncoder`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.ScalarQuantizerEncoder` returns a scalar quantizer encoder System object, `H`. This object maps each input value to a quantization region by comparing the input value to the user-specified boundary points.

```
H =
dsp.ScalarQuantizerEncoder('PropertyName',PropertyValue,...)
```
returns a scalar quantizer encoder object, `H`, with each specified property set to the specified value.

Properties   **BoundaryPointsSource**

        Source of boundary points

        Specify how to determine the boundary points and codebook values as `Property` or `Input port`. The default is `Property`.

Partitioning

        Quantizer is bounded or unbounded

        Specify the quantizer as `Bounded` or `Unbounded`. The default is `Bounded`.

BoundaryPoints

Boundary points of quantizer regions

Specify the boundary points of quantizer regions as a vector. The vector values must be in ascending order. Let [p0 p1 p2 p3 ... pN] denote the boundary points property in the quantizer. If the quantizer is bounded, the object uses this property to specify [p0 p1 p2 p3 ... pN]. If the quantizer is unbounded, the object uses this property to specify [p1 p2 p3 ... p(N-1)] and sets p0 = -Inf and pN = +Inf. This property applies when you set the BoundaryPointsSource property to Property. The default is 1:10. This property is tunable.

SearchMethod

Find quantizer index by linear or binary search

Specify whether to find the appropriate quantizer index using a linear search or a binary search as one of Linear or Binary. The computational cost of the linear search method is of the order P and the computational cost of the binary search method is of the order

$$\log_2(P)$$

where P is the number of boundary points. The default is Linear.

TiebreakerRule

Behavior when input equals boundary point

Specify whether the input value is assigned to the lower indexed region or higher indexed region when the input value equals boundary point by selecting Choose the lower index or Choose the higher index. The default is Choose the lower index.

CodewordOutputPort

Enable output of codeword value

Set this property to true to output the codeword values that correspond to each index value. The default is false.

QuantizationErrorOutputPort

Enable output of quantization error

Set this property to `true` to output the quantization error for each input value. The quantization error is the difference between the input value and the quantized output value. The default is `false`.

Codebook

Codebook

Specify the codebook as a vector of quantized output values that correspond to each region. If the `Partitioning` property is `Bounded` and the boundary points vector has length N, you must set this property to a vector of length N-1. If the `Partitioning` property is `Unbounded` and the boundary points vector has length N, you must set this property to a vector of length N+1. This property applies when you set the `BoundaryPointsSource` property to `Property` and either the `CodewordOutputPort` property or the `QuantizationErrorOutputPort` property is `true`. The default is `1.5:9.5`. This property is tunable.

ClippingStatusOutputPort

Enable output of clipping status

Set this property to `true` to output the clipping status. The output is a 1 when an input value is outside the range defined by the `BoundaryPoints` property. When the value is inside the range, the exception output is a 0. This property applies when you set the `Partitioning` property to `Bounded`. The default is `false`.

OutputIndexDataType

Data type of the index output

Specify the data type of the index output from the object as: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`. The default is `int32`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest` or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

Methods

clone	Create an instance of an object with the same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Operate on inputs to calculate outputs

Examples

Quantize the varying fractional inputs between zero and five to the closest integers, and then plot the results:

```
hsqe = dsp.ScalarQuantizerEncoder;
hsqe.BoundaryPoints = [-.001 .499 1.499 ...
   2.499 3.499 4.499 5.001];
hsqe.CodewordOutputPort = true;
hsqe.Codebook = [0 1 2 3 4 5];
input = (0:0.02:5)';
[index, quantizedValue] = step(hsqe, input);
plot(1:length(input), [input quantizedValue]);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Scalar Quantizer Encoder block reference page. The object properties correspond to the block parameters.

See Also   dsp.ScalarQuantizerDecoder | dsp.VectorQuantizerEncoder

Purpose   Create an instance of an object with the same property values

Syntax

Description   `C = clone(H)` creates a `ScalarQuantizerEncoder` object `C`, with the
same property values as `H`. The clone method creates a new unlocked
object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                       the step method.

                       The getNumInputs method returns a positive integer that is the number
                       of expected inputs (not counting the object itself) to the step method.
                       This value will change if you alter any properties that turn inputs on or
                       off. You must call the step method with the number of input arguments
                       equal to the result of getNumInputs(*H*).

dsp.ScalarQuantizerEncoder.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the ScalarQuantizerEncoder object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.ScalarQuantizerEncoder.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Operate on inputs to calculate outputs

Syntax   INDEX= step(HSQE,INPUT)
[...] = step(HSQE,INPUT,BPOINTS)
[...] = step(HSQE,INPUT,BPOINTS,CODEBOOK)
[...,CODEWORD] = step(HSQE, ...)
[..., QERR] = step(HSQE, ...)
[...,CLIPSTATUS] = step(HSQE, ...)

Description   INDEX= step(HSQE,INPUT) returns the INDEX of the quantization
region to which the INPUT belongs. The input data, boundary points,
codebook values, quantized output values, and the quantization error
must have the same data type whenever they are present.

[...] = step(HSQE,INPUT,BPOINTS) when the
BoundaryPointsSource property is Input port, uses input BPOINTS
as the boundary points .

[...] = step(HSQE,INPUT,BPOINTS,CODEBOOK) uses input BPOINTS
as the boundary points and input CODEBOOK as the codebook when
the BoundaryPointsSource property is Input port and either the
CodewordOutputPort property or the QuantizationErrorOutputPort
property is true.

[...,CODEWORD] = step(HSQE, ...) outputs the CODEWORD values
that corresponds to each index value when the CodewordOutputPort
property is true.

[..., QERR] = step(HSQE, ...) outputs the quantization error QERR
for each input value when the QuantizationErrorOutputPort property
is true.

[...,CLIPSTATUS] = step(HSQE, ...) also returns output
CLIPSTATUS as the clipping status output port for each input
value when the Partitioning property is Bounded and the
ClippingStatusOutputPort property is true. If an input value is
outside the range defined by the BoundaryPoints property, CLIPSTATUS
is true. If an input value is inside the range, CLIPSTATUS is false.

> **Note** The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose   Log simulation data in buffer

Description   The SignalSink object logs MATLAB simulation data. This object accepts any numeric data type.

To log MATLAB simulation data :

1 Define and set up your signal sink. See "Construction" on page 5-951.

2 Call step to log the simulation data according to the properties of dsp.SignalSink. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.SignalSink returns a signal sink, H, that logs 2-D input data in the object.

H = dsp.SignalSink('*PropertyName*',*PropertyValue*,...) returns a signal sink, H, with each specified property set to the specified value.

Properties   **FrameBasedProcessing**

Process input as frames or samples

To enable frame-based processing for 2-D inputs, set this property to true. When you set this property to false, the object performs sample-based processing on 2-D inputs. The signal sink object always performs sample-based processing for N-D inputs when N is greater than 2. The default is true.

BufferLength

Maximum number of input frames or samples to log

Specify the maximum number of frames to log when the object is performing frame-based processing, or the maximum number of samples to log when the object is performing sample-based processing. The object always preserves the most recent data in the buffer. When you specify a buffer length that is greater than the input length, the object pads the end of the logged data with

zeros. To capture all input data without extra padding, set the BufferLength property to inf. The default is inf.

Decimation

Decimation factor

Setting this property to any positive integer d causes the signal sink to write data at every dth sample. The default is 1.

FrameHandlingMode

Output dimensionality for frame-based inputs

Set the dimension of the output array for frame-based inputs as 2-D array (concatenate) or 3-D array (separate). Concatenation occurs along the first dimension for 2-D array (concatenate). This property applies only when you set the FrameBasedProcessing property to true. The default is 2-D array (concatenate).

Buffer

Logged Data (read only)

The signal sink writes simulation data into a buffer. Specify the maximum length of the buffer with the BufferLength property.

Methods		
clone	Create signal logger object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset	Reset internal states of signal logger object
step	Store signal in buffer

Examples

Log input data.

```
hlog = dsp.SignalSink;
 for i=1:10
     y = sin(i);
     step(hlog,y);
 end
log = hlog.Buffer; % log = sin([1;2;3;4;5;6;7;8;9;10])
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Signal To Workspace block reference page. The object properties correspond to the block properties, except:

The object always generates fixed-point output for fixed-point input.

Objects and blocks interpret frames differently. Objects process inputs as frames or as samples by setting the FrameBasedProcessing property. Blocks process inputs as frames or as samples by inheriting the frame information from the input ports. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also

dsp.SignalSource

dsp.SignalSink.clone

Purpose   Create signal logger object with same property values

Syntax   `C = clone(H)`

Description   `C = clone(H)` creates a `SignalSink` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.SignalSink.getNumOutputs

Purpose   Number of outputs of step method

Syntax   `N = getNumOutputs(H)`

Description   `N = getNumOutputs(H)` returns the number of outputs, `N`, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the SignalSink object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.SignalSink.reset

Purpose   Reset internal states of signal logger object

Syntax   reset(H)

Description   reset(H) sets the internal states of the SignalSink object H to their initial values.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.SignalSink.step

Purpose   Store signal in buffer

Syntax   step(H,Y)

Description   step(H,Y) buffers the signal Y. The buffer may be accessed at any time
from the Buffer property of H.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

Purpose Import variable from workspace

Description The SignalSource object imports a variable from the MATLAB workspace.

To import a variable from the MATLAB workspace:

1 Define and set up your signal source. See "Construction" on page 5-961.

2 Call step to import the variable according to the properties of dsp.SignalSource. The behavior of step is specific to each object in the toolbox.

Construction H = dsp.SignalSource returns a signal source System object, H, that outputs the variable one sample or frame at a time.

H = dsp.SignalSource('*PropertyName*',*PropertyValue*,...) returns a signal source object, H, with each specified property set to the specified value.

H = dsp.SignalSource(signal,spf,'*PropertyName*',*PropertyValue*,...) returns a signal source object, H, with the Signal property set to signal, the SamplesPerFrame property set to spf, and other specified properties set to the specified values.

Properties **Signal**

Variable or expression containing the signal

Specify the name of the workspace variable from which to import the signal, or a valid expression specifying the signal. The default is [1:10].

SamplesPerFrame

Number of samples per output frame

Specify the number of samples to buffer into each output frame. This property must be 1 when you specify a 3-D array in the Signal property. The default is 1.

SignalEndAction

Action after final signal values are generated

Specify the output after all of the specified signal samples have been generated as one of Set to zero, Hold final value, or Cyclic repetition. The default is Set to zero.

Methods

clone	Create signal reader object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isDone	End-of-file status for signal reader object
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of signal reader object
step	Read one sample or frame of signal

Examples

Create a signal source to output one sample at a time:

```
hsr1 = dsp.SignalSource;
hsr1.Signal = randn(1024, 1);
y1 = zeros(1024,1);
idx = 1;
```

```
while(~isDone(hsr1))
  y1(idx) = step(hsr1);
  idx = idx+1;
end
```

Create a signal source to output vectors:

```
hsr2 = dsp.SignalSource(randn(1024, 1), 128);
y2 = step(hsr2); % y2 is a 128-by-1 frame of samples
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Signal From Workspace block reference page. The object properties correspond to the block parameters, except:

The object does not have properties that correspond to the **Sample time** or **Warn when frame size does not evenly divide input length** block parameters.

See Also   dsp.SignalSink

dsp.SignalSource.clone

Purpose   Create signal reader object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a SignalSource object C, with the same property values as H. The clone method creates a new unlocked object with un-initialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.SignalSource.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

Purpose   End-of-file status for signal reader object

Syntax   isDone(H)

Description   isDone(H) returns a logical value indicating whether or not the
SignalSource object, H, has reached the end of the imported signal.
If the SignalEndAction property is set to Cyclic repetition, this
method will return true every time the reader reaches the end.

Purpose   Locked status for input attributes and nontunable properties

Syntax   `isLocked(H)`

Description   `isLocked(H)` returns the locked state of the `SignalSource` object `H`.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Reset internal states of signal reader object

Syntax   reset(H)

Description reset(H) resets the signal reader object H, to start reading from the
beginning of the imported signal.

dsp.SignalSource.release

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Read one sample or frame of signal

Syntax   Y = step(H)

Description   Y = step(H) outputs one sample or frame of data, Y, from each column
                  of the imported signal. The imported signal is the variable or expression
                  you specify for the Signal property of the SignalSource System object
                  H.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.SineWave

Purpose   Discrete-time sinusoid

Description   The SineWave object generates a discrete-time sinusoid. The sine wave object generates a real–valued sinusoid or a complex exponential. A real-valued, discrete-time sinusoid is defined as:

$$y(n) = A\sin(2\pi f n + \phi)$$

where A is the amplitude, f is the frequency in hertz, and φ is the initial phase, or phase offset, in radians. A complex exponential is defined as:

$$y(n) = A e^{j(2\pi f n + \phi)}$$

For both real and complex sinusoids, the amplitude, frequency, and phase offsets can be scalars or length-N vectors, where N is the desired number of channels in the output. When you specify at least one of these properties as a length-N vector, scalar values specified for the other properties are applied to each of the N channels.

To generate a discrete-time sinusoid:

1 Define and set up your sine wave. See "Construction" on page 5-972.

2 Call step to generate the sinusoid according to the properties of dsp.SineWave. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.SineWave returns a sine wave object, H, that generates a real-valued sinusoid with an amplitude of 1, a frequency of 100 Hz, and a phase offset of 0. By default, the sine wave object only generates one sample.

H = dsp.SineWave('*PropertyName*',*PropertyValue*, ...) returns a sine wave object, H, with each property set to the specified value.

H = dsp.SineWave(AMP,FREQ,PHASE,
'*PropertyName*',*PropertyValue*, ...) returns a sine wave object, H, with the Amplitude property set to AMP, the Frequency property set to

FREQ, the `PhaseOffset` property set to PHASE, and the other specified properties set to the specified values.

Properties

Amplitude

Amplitude of the sine wave

Specify the amplitude as a length-*N* vector containing the amplitudes of the sine waves in each of *N* output channels, or a scalar to apply to all *N* channels. The vector length must equal that specified for the `Frequency` and `PhaseOffset` properties. The default value is 1. This property is tunable when the `Method` property is `Trigonometric function` or `Differential`.

Frequency

Frequency of the sine wave

Specify a length-*N* vector containing frequencies, in hertz, of the sine waves in each of *N* output channels, or a scalar to apply to all *N* channels. The vector length must equal that specified for the `Amplitude` and `PhaseOffset` properties. You can specify positive, zero, or negative frequencies. The default is 100. This property is tunable when the `Method` property is `Trigonometric function`.

PhaseOffset

Phase offset of the sine wave in radians

A length-*N* vector containing the phase offsets, in radians, of the sine waves in each of *N* output channels, or a scalar to apply to all *N* channels. The vector length must equal that specified for the `Amplitude` and `Frequency` properties. The default value is 0. This property is tunable when the `Method` property is `Trigonometric function`.

ComplexOutput

Indicates whether the sine wave is complex or real

Set to `true` to output a complex exponential. The default value is `false`.

dsp.SineWave

Method

Method used to generate sinusoids

The sinusoids are generated by either the `Trigonometric function` , `Table lookup`, or `Differential` methods . The `trigonometric function` method computes the sinusoid by sampling the continuous-time sinusoid. The `table lookup` method precomputes the unique samples of every output sinusoid at the start of the simulation, and recalls the samples from memory as needed. The `differential` method uses an incremental algorithm. This algorithm computes the output samples based on the output values computed at the previous sample time and precomputed update terms. The default value is `Trigonometric function`.

TableOptimization

Optimizes the table of sine values for speed or memory

Optimizes the table of sine values for `Speed` or `Memory`. When optimized for speed, the table contains k elements, and when optimized for memory, the table contains $k/4$ elements, where k is the number of input samples in one full period of the sine wave. This property applies only when the `Method` property is `Table lookup`. The default value is `Speed`.

SampleRate

Sampling rate for the sine wave

Specify the sampling rate of the output, in hertz, as a positive numeric scalar. The default is 1000.

SamplesPerFrame

Number of samples per frame

Specify the number of consecutive samples from each sinusoid to buffer into the output frame. The default is 1.

OutputDataType

Output data type

Specify the output data type as double, single, or Custom. The default value is double.

Fixed-Point Properties

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a numerictype object with a Signedness of Auto. This property applies only when the OutputDataType property is Custom. The default value is numerictype([]1,16).

Methods		
	clone	Create sine wave object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	reset	Reset sine wave to the beginning
	step	Discrete-time sine wave

Examples   Generate a sine wave with an amplitude of 2, frequency of 10 Hz, and initial phase of 0:

```
hsin1 = dsp.SineWave(2, 10);
hsin1.SamplesPerFrame = 1000;
y = step(hsin1);
```

```
plot(y);
```

Generate two sine waves offset by a phase of π/2 radians:

```
hsin2 = dsp.SineWave;
hsin2.Frequency = 10;
hsin2.PhaseOffset = [0 pi/2];
hsin2.SamplesPerFrame = 1000;
y = step(hsin2);
plot(y);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Sine Wave block reference page. The object properties correspond to the block parameters.

See Also   dsp.Chirp | dsp.NCO

Purpose   Create sine wave object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a sine wave object, C, with the same property
                   values as H. The clone method creates a new unlocked object with
                   uninitialized states.

dsp.SineWave.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                  the step method.

                  The getNumInputs method returns a positive integer that is the number
                  of expected inputs (not counting the object itself) to the step method.
                  This value will change if you alter any properties that turn inputs on or
                  off. You must call the step method with the number of input arguments
                  equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step
method.

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the sine wave object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.SineWave.reset

Purpose　　Reset sine wave to the beginning

Syntax　　`reset(H)`

Description　　`reset(H)` resets the sine wave object to the beginning (*n=0*). If you invoke the `step` method without first invoking the `reset` method, the sine wave object generates additional samples of the original sinusoid. As an example:

```
H = dsp.SineWave('Amplitude',1,'Frequency',1/4,'SamplesPerFrame',7);
% 7 samples of a sine wave with frequency of 1/4 Hz
y = step(H);
% Call step method without reset
y1 = step(H);
stem([y; y1])
% y1 is a continuation of y
% Now reset
reset(H);
y2 = step(H);
% y2 starts at n=0 and equals y
isequal(y,y2)
```

Purpose   Discrete-time sine wave

Syntax   `Y = step(H)`

Description   `Y = step(H)` produces the sine wave `Y` using the specifications in the object `H`.

dsp.SpectrumAnalyzer

Purpose   Display frequency spectrum of time-domain signals

Description   The `SpectrumAnalyzer` object displays the frequency spectrum of time-domain signals.

To display the spectra of signals in the Spectrum Analyzer:

1 Define and set up your Spectrum Analyzer. See "Construction" on page 5-986.

2 Call `step` to display the frequency spectrum of the signals in the Spectrum Analyzer figure. The behavior of `step` is specific to each object in the toolbox.

Use the MATLAB `clear` function to close the Spectrum Analyzer figure window and clear its associated data. Use the `hide` method to hide the Spectrum Analyzer window and the `show` method to make it visible.

See the following sections for more information on the Spectrum Analyzer Graphical User Interface:

dsp.SpectrumAnalyzer

Construction   H = dsp.SpectrumAnalyzer creates a Spectrum Analyzer System object, H. This object displays the frequency spectrum of real- and complex-valued floating- and fixed-point signals.

H = dsp.SpectrumAnalyzer('*Name*','*Value*',...) creates a Spectrum Analyzer System object, H, with each specified property *Name* set to the specified value. You can specify *Name–Value* arguments in any order.

Properties   **CenterFrequency**

Frequency over which span is centered

Specify as a real scalar the center frequency, in hertz, of the span over which the Spectrum Analyzer computes and plots the spectrum. This property applies when you set the FrequencySpan property to 'Span and center frequency'. The overall span, defined by the Span and CenterFrequency properties, must fall within the Nyquist frequency interval. When the TwoSidedSpectrum property is set to true, the interval is

$$\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ hertz. When}$$

the TwoSidedSpectrum property is set to false, the interval

$$\text{is} \left[0, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ hertz.}$$

This property is Tunable.

Default: 0

FFTLength

FFT length

Specify as a positive, scalar integer the length of the FFT that Spectrum Analyzer uses to compute spectral estimates. This property applies only when you set the FFTLengthSource property to 'Property'.

This property is Tunable.

Default: 128

FFTLengthSource

Source of the FFT length value

Specify the source of the FFT length value as one of 'Auto' or 'Property'.

- When you set this property to 'Auto', the Spectrum Analyzer sets the FFT length to the data segment length required to achieve the specified resolution bandwidth.

- When you set this property to 'Property', specify the number of FFT points using the FFTLength property.

This property is Tunable.

Default: 'Auto'

FrequencyOffset

Frequency offset

Specify as a real, scalar value a frequency offset, in hertz. The *frequency*-axis values, which you also specify, are offset by the value specified in this property. The overall span must fall within the *Nyquist* frequency interval. When the TwoSidedSpectrum property is true, the Nyquist interval is

$$\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset$$ hertz. When you set the TwoSidedSpectrum property to false, the Nyquist

interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz. You can control the overall span in different ways based on how you set the FrequencySpan property.

Default: 0

FrequencyScale

Frequency scale

Specify the frequency scale as either `'Linear'` or `'Log'`. When you set this property to `'Log'`, the Spectrum Analyzer displays the frequencies on the *x*-axis on a logarithmic scale. To use the `'Log'` setting, you must also set the `TwoSidedSpectrum` property to `false`. When the `TwoSidedSpectrum` property is `true`, you must set this property to `'Linear'`.

This property is Tunable.

Default: `'Linear'`

FrequencySpan

Frequency span mode

Specify the frequency span mode as one of `'Full'`, `'Span and center frequency'`, or `'Start and stop frequencies'`.

- When you set this property to `'Full'`, the Spectrum Analyzer computes and plots the spectrum over the entire *Nyquist* frequency interval. When the `TwoSidedSpectrum` property is `true`, the Nyquist interval is

$$\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset \text{ hertz. If}$$

you set the `TwoSidedSpectrum` property to `false`, the Nyquist

interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

- When you set this property to `'Span and center frequency'`, the Spectrum Analyzer computes and plots the spectrum over the interval specified by the `Span` and `CenterFrequency` properties.

- When you set this property to `'Start and stop frequencies'`, the Spectrum Analyzer computes and plots the

spectrum over the interval specified by the `StartFrequency` and `StopFrequency` properties.

This property is Tunable.

Default: `'Full'`

Grid

Option to enable or disable grid display

When you set this property to `true`, the grid appears. When you set this property to `false`, the grid does not appear.

This property is Tunable.

Default: `false`

LegendSource

Source of legend

Specify the source of the legend as either `None` or `Auto`. When you set this property to `'None'`, the Spectrum Analyzer does not display a legend. When you set this property to `'Auto'`, the Spectrum Analyzer displays a legend with automatic string labels for each input channel.

This property is Tunable.

Default: `'None'`

MaxHoldTrace

Max-hold trace flag

Set this property to `true` to compute and plot the maximum-hold spectrum of each input channel. The maximum-hold spectrum at each frequency bin is computed by keeping the maximum value of all the power spectrum estimates. When you toggle this property, the Spectrum Analyzer resets its maximum-hold computations.

This property is Tunable.

See also: `MinHoldTrace`, `NormalTrace`.

Default: `false`

MinHoldTrace

Min-hold trace flag

Set this property to `true` to compute and plot the minimum-hold spectrum of each input channel. The minimum-hold spectrum at each frequency bin is computed by keeping the minimum value of all the power spectrum estimates. When you toggle this property, the Spectrum Analyzer resets its minimum-hold computations.

This property is Tunable.

See also: `MaxHoldTrace`, `NormalTrace`.

Default: `false`

Name

Caption to display on Spectrum Analyzer window

Specify the caption to display on the Spectrum Analyzer window as any string.

This property is Tunable.

Default: `'Spectrum Analyzer'`

NormalTrace

Normal trace flag

Set this property to `false` to remove the display of the normal traces. These traces display the free-running spectral estimates. Note that even when the traces are removed from the display, the Spectrum Analyzer continues its spectral computations.

This property is Tunable.

See also: `MaxHoldTrace`, `MinHoldTrace`.

Default: `true`

Position

Spectrum Analyzer window position in pixels

Specify, in pixels, the size and location of the Spectrum Analyzer window as a 4-element double vector of the form, `[left bottom width height]`. You can place the Spectrum Analyzer window in a specific position on your screen by modifying the values for this property.

This property is Tunable.

Default: The default depends on your screen resolution. By default, the Spectrum Analyzer window appears in the center of your screen with a width of `410` pixels and height of `300` pixels.

PowerUnits

Power units

Specify the units in which the Spectrum Analyzer displays power values as either `'dBm'`, `'dBW'`, or `'Watts'`.

This property is Tunable.

Default: `'dBm'`

RBW

Resolution bandwidth

Specify as a real, positive scalar the resolution bandwidth (RBW), in hertz. RBW controls the spectral resolution of Spectrum Analyzer. This property applies only when you set the `RBWSource` property to `'Property'`. You must specify a value to ensure that

there are at least two RBW intervals over the specified frequency span. Thus, the ratio of the overall span to RBW must be greater

than two: $\frac{span}{RBW} > 2$. You can specify the overall span in different ways based on how you set the FrequencySpan property.

This property is Tunable.

See also: RBWSource.

Default: 9.76

RBWSource

Source of resolution bandwidth value

Specify the source of the resolution bandwidth (RBW) as either 'Auto' or 'Property'.

- When you set this property to 'Auto', the Spectrum Analyzer adjusts the spectral estimation resolution. This adjustment ensures that there are 1024 RBW intervals over the defined frequency span.

- When you set this property to 'Property', specify the resolution bandwidth directly when you set the RBW property.

This property is Tunable.

See also: RBW.

Default: 'Auto'

ReferenceLoad

Reference load

Specify as a real, positive scalar the load, in ohms, that the Spectrum Analyzer uses as a reference to compute power values.

This property is Tunable.

Default: 1

SampleRate

Sample rate of input

Specify the sampling rate, in hertz, of the input signals, as a finite numeric scalar.

Default: 10e3

SidelobeAttenuation

Sidelobe attenuation of window

Specify as a real, positive scalar the window sidelobe attenuation, in decibels (dB). This property applies only when you set the Window property to 'Chebyshev' or 'Kaiser'.

This property is Tunable.

Default: 60

Span

Frequency span over which spectrum is computed and plotted

Specify as a real, positive scalar the frequency span, in hertz, over which the Spectrum Analyzer computes and plots the spectrum. This property applies only when you set the FrequencySpan property to 'Span and center frequency'. The overall span, defined by this property and the CenterFrequency property, must fall within the *Nyquist* frequency interval. When the TwoSidedSpectrum property is true, the Nyquist interval is

$$\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2}\right] + FrequencyOffset \text{ hertz. When}$$

you set the TwoSidedSpectrum property to false, the Nyquist

interval is $\left[0, \frac{SampleRate}{2}\right] + FrequencyOffset$ hertz.

This property is Tunable.

See also: `CenterFrequency`, `TwoSidedSpectrum`.

Default: `10e3`

SpectralAverages

Number of spectral averages

Specify as a positive, scalar integer the number of spectral averages. The Spectrum Analyzer computes the current power spectrum estimate by averaging the last *N* power spectrum estimates. This property defines the number of spectral averages, *N*.

This property is Tunable.

Default: `10`

SpectrumType

Spectrum type

Specify the spectrum type as one of `'Power'` or `'Power density'`.

- When you set this property to `'Power'`, the Spectrum Analyzer shows the power spectrum.

- When you set this property to `'Power density'`, the Spectrum Analyzer shows the power spectral density. The power spectral density is the magnitude squared of the spectrum normalized to a bandwidth of 1 hertz.

This property is Tunable.

Default: `'Power'`

StartFrequency

Start frequency over which spectrum is computed

Specify as a real scalar the start frequency, in hertz, over which the Spectrum Analyzer computes and plots the spectrum. This property applies only when you set the FrequencySpan property to 'Start and stop frequencies'. The overall span, which is defined by StopFrequency and this property, must fall within the *Nyquist* frequency interval. When the TwoSidedSpectrum property is true, the Nyquist interval is

$$\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ hertz. If you}$$

set the TwoSidedSpectrum property to false, the Nyquist interval

$$\text{is} \left[0, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ hertz.}$$

This property is Tunable.

See also: StopFrequency, TwoSidedSpectrum.

Default: -5e3

StopFrequency

Stop frequency over which spectrum is computed

Specify as a real scalar the stop frequency, in hertz, over which the Spectrum Analyzer computes and plots the spectrum. This property applies only when you set the FrequencySpan property to 'Start and stop frequencies'. The overall span, defined by this property and the StartFrequency property, must fall within the *Nyquist* frequency interval. When the TwoSidedSpectrum property is set to true, the interval is

$$\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ hertz. When}$$

the TwoSidedSpectrum property is set to false, the interval

$$\text{is} \left[0, \frac{SampleRate}{2} \right] + FrequencyOffset \text{ hertz.}$$

This property is Tunable.

See also: `StartFrequency`, `TwoSidedSpectrum`.

Default: 5e3

Title

Display title

Specify the display title as a string.

This property is Tunable.

Default: `''`

TwoSidedSpectrum

Two sided spectrum flag

Set this property to `true` to compute and plot two-sided spectral estimates. Set this property to `false` to compute and plot one-sided spectral estimates. If you set this property to `false`, then the input signal must be real valued. When the input signal is complex valued, you must set this property to `true`.

When this property is `false`, Spectrum Analyzer uses *power folding*. The *y*-axis values are twice the amplitude that they would be if this property were set to `true`, except at 0 and the Nyquist frequency. A one-sided PSD contains the total power of the signal in the frequency interval from DC to half of the Nyquist rate. For more information, see the `pwelch` function reference page.

Default: `true`

Window

Window function

Specify a window function for the spectral estimator as one of the options in the following table. For information on any window

function, follow the link to the corresponding function reference in the Signal Processing Toolbox documentation.

Window Option	Corresponding Function in Signal Processing Toolbox
'Rectangular'	rectwin
'Chebyshev'	chebwin
'Flat Top'	flattopwin
'Hamming'	hamming
'Hann'	hann
'Kaiser'	kaiser

This property is Tunable.

Default: 'Hann'

YLabel

The label for the *y*-axis

Specify as a string the *y*-axis label. Regardless of this property, Spectrum Analyzer always displays power units as one of 'dBm', 'dBW', 'Watts', 'dBm/Hz', 'dBW/Hz', 'Watts/Hz'.

This property is Tunable.

See also: PowerUnits.

Default: ''

YLimits

The limits for the *y*-axis

Specify as a 2-element numeric vector the *y*-axis limits, [ymin, ymax]. The units directly depend upon the PowerUnits property.

This property is Tunable.

dsp.SpectrumAnalyzer

Default: `[-80, 20]`

Methods

clone	Create spectrum scope object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
hide	Hide Spectrum Analyzer window
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of Spectrum Analyzer object
show	Make Spectrum Analyzer window visible
step	Display signal in Spectrum Analyzer figure

Examples

View a two-sided power spectrum of a sine wave with noise on the Spectrum Analyzer.

```
hsin = dsp.SineWave('Frequency',100,'SampleRate',1000);
hsin.SamplesPerFrame = 1000;
hsa = dsp.SpectrumAnalyzer('SampleRate',hsin.SampleRate);
for ii = 1:250
  x = step(hsin) + 0.05*randn(1000,1);
  step(hsa, x);
end
```

Run the `release` method to automatically scale the axes.

```
release(hsa);
```

Run the MATLAB clear function to close the Spectrum Analyzer window.

```
clear('hsa');
```

View a one-sided power spectrum made from the sum of five real sine waves with different amplitudes and frequencies.

```
Fs = 100e6;  % Sampling frequency
fSz = 5000;  % Frame size

hsin1 = dsp.SineWave(1e0,   5e6, 0, 'SamplesPerFrame', fSz, 'SampleRat
hsin2 = dsp.SineWave(1e-1, 15e6, 0, 'SamplesPerFrame', fSz, 'SampleRat
hsin3 = dsp.SineWave(1e-2, 25e6, 0, 'SamplesPerFrame', fSz, 'SampleRat
hsin4 = dsp.SineWave(1e-3, 35e6, 0, 'SamplesPerFrame', fSz, 'SampleRat
hsin5 = dsp.SineWave(1e-4, 45e6, 0, 'SamplesPerFrame', fSz, 'SampleRat

hsb = dsp.SpectrumAnalyzer;
hsb.SampleRate = Fs;
hsb.SpectralAverages = 1;
hsb.TwoSidedSpectrum = false;
hsb.RBWSource = 'Auto';
hsb.PowerUnits = 'dBW';
hsb.Position = [749 227 976 721];
for idx = 1:1e2
    y1 = step(hsin1);
    y2 = step(hsin2);
    y3 = step(hsin3);
    y4 = step(hsin4);
    y5 = step(hsin5);
    step(hsb,y1+y2+y3+y4+y5+0.0001*randn(fSz,1));
end
```

Run the release method to automatically scale the axes.

```
release(hsb);
```

Run the MATLAB clear function to close the Spectrum Analyzer window.

```
clear('hsb');
```

Signal Display

The Spectrum Analyzer indicates the spectrum computation settings that are represented in the current display. Check the **Resolution Bandwidth**, **Number of FFT Points**, **Frequency Span**, and **Center Frequency** indicators on the axes in the scope window for this information. These indicators relate to the **Minimum Frequency-Axis limit** and **Maximum Frequency-Axis limit** values on the *frequency*-axis of the scope window. The values specified by these indicators may be changed by modifying parameters in the **Spectrum Settings** panel. You can also view the object state and the amount of time data that correspond to the current display. Check the **Simulation Status** and **Simulation time** indicators on the status bar in the scope window for this information. The following figure highlights these aspects of the Spectrum Analyzer window.

- **Frequency Span** — The range of values shown on the *frequency*-axis on the Spectrum Analyzer window.

  Details

  Spectrum Analyzer sets the frequency span using the values of parameters on the **Main options** pane of the **Spectrum Settings** panel.

  - Span (Hz) and CF (Hz) visible — The **Frequency Span** value equals the **Span** parameter in the **Main options** pane. You can also get or set this value from the Span property when the FrequencySpan property is set to 'Span and Center Frequency'.

  - FStart (Hz) and FStop (Hz) — The **Frequency Span** value equals the difference of the **FStop** and **FStart** parameters in the

    Main options pane, as given by the formula: $f_{span} = f_{stop} - f_{start}$. You can also get or set these values from the StartFrequency and StopFrequency properties when the FrequencySpan property is set to 'Start and stop frequencies'.

  By default, the **Full Span** check box in the **Main options** pane is enabled, and its equivalent FrequencySpan property is set to 'Full'. In this case, the Spectrum Analyzer computes and plots the spectrum over the entire *Nyquist* frequency interval. When the **Two-sided spectrum** check box in the **Trace options** pane is enabled, and its equivalent TwoSidedSpectrum property is true, the Nyquist interval

  is $\left[-\dfrac{SampleRate}{2}, \dfrac{SampleRate}{2} \right] + FrequencyOffset$ hertz. If you set the TwoSidedSpectrum property to false, the Nyquist interval

  is $\left[0, \dfrac{SampleRate}{2} \right] + FrequencyOffset$ hertz.

- **Resolution Bandwidth** — The smallest positive frequency that can be resolved.

Details

Spectrum Analyzer sets the resolution bandwidth using the value of the **RBW** parameter on the **Main options** pane of the **Spectrum Settings** panel. You can also get or set this value from the RBW property when RBWSource is set to 'Property'. By default, the **RBW** parameter on the **Main options** pane and the related RBWSource property are set to 'Auto'. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified **Frequency Span**.

You can set the resolution bandwidth to whatever value you choose. For this reason, there is a minimum boundary on the number of input samples required to compute a spectral update. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly

related to *RBW* by the following equation: $N = \dfrac{0.2 \times NENBW \times F_s}{RBW}$. *NENBW* is the normalized effective noise bandwidth, a factor of the windowing method used, which is shown in the **Frequency Options** pane. F_s is the sample rate. In some cases, the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer produces a message on the display, as shown in the following figure.

Spectrum Analyzer removes this message and displays a spectral estimate as soon as enough data has been input.

- **Number of FFT Points** — The Fast Fourier Transform (FFT) length.

Details

Spectrum Analyzer sets the number of FFT points using the value of the **NFFT** parameter on the **Trace options** pane of the **Spectrum Settings** panel. You can also get or set this value from the FFTLength property when the FFTLengthSource property is set to 'Property'. By default, the **NFFT** parameter on the **Trace options** pane and the related FFTLengthSource property are set to 'Auto'. In this case, the Spectrum Analyzer sets the number of FFT points to the data segment length needed to achieve the specified **Resolution Bandwidth**.

- **Center Frequency** — The value shown at the middle point of the *frequency*-axis on the Spectrum Analyzer display.

  Details

  Spectrum Analyzer sets the center frequency using the values of parameters on the **Main options** pane of the **Spectrum Settings** panel.

  - Span (Hz) and CF (Hz) visible — The **Center Frequency** value equals the **CF** parameter in the **Main options** pane. You can also get or set this value from the CenterFrequency property when the FrequencySpan property is set to 'Span and Center Frequency'.

  - FStart (Hz) and FStop (Hz) visible — The **Center Frequency** value equals half the sum of the **FStop** and **FStart** parameters in the **Main options** pane, as given by the

    formula: $f_{center} = \dfrac{f_{start} + f_{stop}}{2}$. You can also get or set these values from the StartFrequency and StopFrequency properties when the FrequencySpan property is set to 'Start and stop frequencies'.

- **Minimum Frequency-Axis limit** — The value shown at the far-left side of the frequency-axis on the Spectrum Analyzer window. Spectrum Analyzer sets the minimum *frequency*-axis limit using the values of the parameters on the **Main options** pane of the **Spectrum Settings** panel.

  Details

  - Span (Hz) and CF (Hz) visible — The **Minimum Frequency-Axis limit** value equals the difference between the center frequency and half the frequency span, as given by the

    following formula: $f_{min} = f_{center} - \dfrac{f_{span}}{2}$. In this case, you can modify the minimum *frequency*-axis limit by changing the **Span** and **CF** parameters in the **Main options** pane. You can also get

or set these values from the Span and CenterFrequency properties when the FrequencySpan property is set to 'Span and Center Frequency'.

- FStart (Hz) and FStop (Hz) visible — The **Minimum Frequency-Axis limit** value equals the **FStart** parameter in the **Main options** pane. You can also get or set this value from the StartFrequency property when the FrequencySpan property is set to 'Start and stop frequencies'.

- **Maximum Frequency-Axis limit** — The value shown at the far-right side of the *frequency*-axis on the Spectrum Analyzer window. Spectrum Analyzer sets the maximum *frequency*-axis limit using the values of the parameters on the **Main options** pane of the **Spectrum Settings** panel.

  Details

  - Span (Hz) and CF (Hz) visible — The **Maximum Frequency-Axis limit** value equals the sum of the center frequency and half the frequency span, as given by the following

    formula: $f_{max} = f_{center} + \dfrac{f_{span}}{2}$. In this case, you can modify the maximum *frequency*-axis limit by changing the **Span** and **CF** parameters in the **Main options** pane. You can also get or set these values from the Span and CenterFrequency properties when the FrequencySpan property is set to 'Span and Center Frequency'.

  - FStart (Hz) and FStop (Hz) visible — The **Maximum Frequency-Axis limit** value equals the **FStop** parameter in the **Main options** pane. You can also get or set this value from the StopFrequency property when the FrequencySpan property is set to 'Start and stop frequencies'.

- **Simulation Status** — Provides the current status of the model simulation.

Details

The status can be one of the following conditions:

- Processing — Occurs after you construct the SpectrumAnalyzer
  object and before you run the release method.

- Stopped — Occurs after you run the release method.

The **Simulation Status** is part of the **Status Bar** in the Spectrum
Analyzer window. You can choose to hide or display the entire **Status
Bar**. From the Spectrum Analyzer menu, select **View > Status Bar**.

- **Simulation time** — The amount of time that has progressed since
  the Spectrum Analyzer System object was first constructed or reset.

Details

Every time you call the step method, the simulation time increases
by the number of rows in the input signal divided by the sample

rate, as given by the following formula: $t_{sim} = t_{sim} + \dfrac{\text{length}(x)}{SampleRate}$.
At the beginning of a simulation, you can modify the **SampleRate**
parameter on the **Main options** pane of the **Spectrum Settings**
panel. You can also set the sample rate using the SampleRate
property.

The **Simulation time** indicator is a component of the **Status Bar** in
the Spectrum Analyzer window. You can choose to hide or display
the entire **Status Bar**. From the Spectrum Analyzer menu, select
View > Status Bar .

For more information, see "Spectrum Settings" on page 5-1009.

Toolbar   The Spectrum Analyzer toolbar contains the following buttons.

Print, Settings, and Properties Buttons

Button	Menu Location	Shortcut Keys	Description
🖶	**File > Print**	**Ctrl+P**	Print the current Spectrum Analyzer window. To enable printing, run the release method.
			To print the current scope window to a figure rather than sending it to your printer, select **File > Print to figure**.
📊	**View > Spectrum Settings**	N/A	Open or close the **Spectrum Settings** panel. You can modify the settings in this panel to control the manner in which the spectrum is calculated.
			See the "Spectrum Settings" on page 5-1009 section for more information.
⚙	**View > Properties**	N/A	Open the **Visuals — Spectrum Options** dialog box.
			See the "Visuals — Spectrum Options" on page 5-1022 section for more information.

Axes Control Buttons

Button	Menu Location	Shortcut Keys	Description
🔍	**Tools > Zoom In**	N/A	When this tool is active, you can zoom in on the scope window. To do so, click in the center of your area of interest, or click and drag your cursor to draw

			a rectangular area of interest inside the scope window.
⼮Q⼮	**Tools > Zoom X**	N/A	When this tool is active, you can zoom in on the *x*-axis. To do so, click inside the scope window, or click and drag your cursor along the *x*-axis over your area of interest.
⼮Q⼮	**Tools > Zoom Y**	N/A	When this tool is active, you can zoom in on the *y*-axis. To do so, click inside the scope window, or click and drag your cursor along the *y*-axis over your area of interest.
⛶	**Tools > Scale Axes Limits**	**Ctrl+A**	Click this button to scale the axes in the active scope window. Alternatively, you can enable automatic axes scaling by selecting one of the following options from the **Tools** menu: • **Automatically Scale Axes Limits** — When you select this option, the scope scales the axes as needed during simulation. • **Scale Axes Limits at Stop** — When you select this option, the scope scales the axes each time the simulation is stopped.

Measurements Buttons

	Tools > Measurements > Peak Finder	N/A	Open or close the **Peak Finder** panel. This panel displays maxima and the frequencies at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified.
			See the "Peak Finder Panel" on page 5-1018 section for more information.

Note The zoom buttons do not change the settings related to span for Spectrum Analyzer. These buttons are purely graphical. Spectrum computations are not affected when you zoom.

You can control whether this toolbar appears in the Spectrum Analyzer window. From the Spectrum Analyzer menu, select **View > Toolbar**.

Spectrum Settings

The **Spectrum Settings** panel appears at the right side of the Spectrum Analyzer figure. This panel enables you to modify settings to control the manner in which the spectrum is calculated. You can choose to hide or display the **Spectrum Settings** panel. In the Spectrum Analyzer menu, select **View > Spectrum Settings**. Alternatively, in the Spectrum Analyzer toolbar, select the Spectrum Settings button.

The **Spectrum Settings** panel is separated into three panes, labeled **Main Options**, **Trace Options**, and **Frequency Options**. You can expand each pane to see the available options.

Main Options Pane

The **Main Options** pane enables you to modify the main options.

- **Sample rate (Hz)** — The sample rate, in hertz, of the input signals. This parameter is also known as sampling frequency.

- **Full span** — Enable this check box to have Spectrum Analyzer compute and plot the spectrum over the entire *Nyquist* frequency interval. By default, when the **Two-sided spectrum** check box is also enabled, the Nyquist interval is

$$\left[-\frac{SampleRate}{2}, \frac{SampleRate}{2} \right] + FrequencyOffset$$ hertz. If you clear the **Two-sided spectrum** check box, the Nyquist interval

is $\left[0, \frac{SampleRate}{2} \right] + FrequencyOffset$ hertz. Enabling this check box is equivalent to setting the FrequencySpan property to 'Full'.

- **Span (Hz)** and **CF (Hz)**, or **FStart (Hz)** and **FStop (Hz)** — When **Span (Hz)** is showing in the **Main Options** pane, you define the range of values shown on the *frequency*-axis on the Spectrum Analyzer window using frequency span and center frequency. From the drop-down list, select **FStart (Hz)** to define the range of *frequency*-axis values using start frequency and stop frequency instead.

  - **Span (Hz)** — The frequency span, in hertz. This parameter defines the range of values shown on the *frequency*-axis on the Spectrum Analyzer window. When this parameter is set to a numeric value, then it is equivalent to the Span property when the FrequencySpan property is set to 'Span and Center Frequency'.

    When this parameter is set to

  - **CF (Hz)** — The center frequency, in hertz. This parameter defines the value shown at the middle point of the *frequency*-axis on the Spectrum Analyzer window. This parameter is equivalent to the CenterFrequency property when the FrequencySpan property is set to 'Span and Center Frequency'.

  - **FStart (Hz)** — The start frequency, in hertz. This parameter defines the value shown at the leftmost side of the *frequency*-axis on the Spectrum Analyzer window. This parameter is equivalent to the StartFrequency property when the FrequencySpan property is set to 'Start and stop frequencies'.

  - **FStop (Hz)** — The stop frequency, in hertz. The parameter defines the value shown at the rightmost side of the *frequency*-axis on the Spectrum Analyzer window. This parameter is equivalent to the StopFrequency property when the FrequencySpan property is set to 'Start and stop frequencies'.

- **RBW (Hz)** — The resolution bandwidth, in hertz. This parameter defines the smallest positive frequency that can be resolved. By default, this parameter is set to Auto. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 *RBW* intervals over the specified **Frequency Span**. This case is equivalent to when you set the RBWSource property to 'Auto'.

If you set this parameter to a numeric value, then you must specify a value that ensures there are at least two *RBW* intervals over the specified frequency span. In other words, the ratio of the overall span to *RBW* must be at least two: $\frac{span}{RBW} > 2$. In this case, this parameter is equivalent to the RBW property, when RBWSource is set to 'Property'.

- **Samples/update** — The number of input samples required to compute one spectral update. This parameter is directly related to *RBW* by the following equation: $N = \frac{0.2 \times NENBW \times F_s}{RBW}$. *NENBW* is the normalized effective noise bandwidth, a factor of the windowing method used, which is shown in the **Frequency Options** pane. F_s is the sample rate. If the number of samples provided in the input are not sufficient to achieve the resolution bandwidth that you specify, Spectrum Analyzer produces a message on the display as shown in the following figure.

Trace Options Pane

The **Trace Options** pane enables you to modify the trace options.

- **Type** — The type of spectrum to display. Acceptable options are
  Power and Power density. When you set this parameter to Power,
  the Spectrum Analyzer shows the power spectrum. When you set
  this parameter to Power density, the Spectrum Analyzer shows the
  power spectral density. The power spectral density is the magnitude
  of the spectrum normalized to a bandwidth of 1 hertz. This parameter
  is equivalent to the SpectrumType property.

- **Units** — The units of the spectrum. Acceptable options are dBm, dBW,
  and Watts. This parameter is equivalent to the PowerUnits property.

- **Averages** — Specify as a positive, scalar integer the number
  of spectral averages. Spectrum Analyzer computes the current
  power spectrum estimate by averaging the last N power spectrum

estimates. This property defines the number of spectral averages, *N*. This parameter is equivalent to the `SpectralAverages` property.

- **NFFT** — The number of Fast Fourier Transform (FFT) points. This parameter defines the length of the FFT that Spectrum Analyzer uses to compute spectral estimates. Acceptable options are `Auto` or a positive, scalar integer. By default, when this parameter is set to `Auto`, Spectrum Analyzer sets the number of FFT points to the data segment length needed to achieve the resolution bandwidth you specified. This case is equivalent to when you set the `FFTLengthSource` property to `'Auto'`.

  When this parameter is set to a positive integer, this parameter is equivalent to the `n` parameter that you can set when you run the MATLAB `fft` function. In this case, this parameter is also equivalent to `FFTLength` when you set `FFTLengthSource` to `'Property'`.

- **Ref. load** — The reference load, in ohms, used to scale the spectrum. Specify as a real, positive scalar load, in ohms, that the Spectrum Analyzer uses as a reference to compute power values. This parameter is equivalent to the `ReferenceLoad` property.

- **Normal trace** — Normal trace view. By default, when this check box is enabled, Spectrum Analyzer calculates and plots the power spectrum or power spectrum density computed by the *Welch* estimator. The Welch estimator performs a smoothing operation by averaging a number of spectral estimates. To clear this check box, you must first select either the **Max hold trace** or the **Min hold trace** check box. This parameter is equivalent to the `NormalTrace` property.

- **Max hold trace** — Maximum hold trace view. Select this check box to enable Spectrum Analyzer to plot the maximum spectral values of all the estimates obtained. This parameter is equivalent to the `MaxHoldTrace` property.

- **Min hold trace** — Minimum hold trace view. Select this check box to enable Spectrum Analyzer to plot the minimum spectral values of all the estimates obtained. This parameter is equivalent to the `MinHoldTrace` property.

• **Two-sided spectrum** — Select this check box to enable two-sided spectrum view. In this view, both negative and positive frequencies are shown. If you clear this check box, Spectrum Analyzer shows a one-sided spectrum with only positive frequencies. Spectrum Analyzer requires that this parameter is selected when the input signal is complex-valued. This parameter is equivalent to the `TwoSidedSpectrum` property.

Frequency Options Pane

The **Frequency Options** pane enables you to modify the frequency options.

• **Scale** — Linear or logarithmic scale. When the frequency span contains negative frequency values, Spectrum Analyzer disables the logarithmic option. This parameter is equivalent to the `FrequencyScale` property.

• **Offset** — The constant frequency offset to apply to the entire spectrum. This constant offset parameter is simply added to the values on the *frequency*-axis in the Spectrum Analyzer window. It is not used in any spectral computations. It does not affect any settings related to frequency span or resolution. This parameter is equivalent to the `FrequencyOffset` property.

• **Window** — The windowing method to apply to the spectrum. This parameter is equivalent to the `Window` property.

Windowing is used to control the effect of sidelobes in spectral estimation. For more information about windowing, see "Windows" in the Signal Processing Toolbox documentation.

• **Attenuation (dB)** — The sidelobe attenuation, in decibels (dB). This property applies only when you set the **Window** parameter to `Chebyshev` or `Kaiser`. This parameter is equivalent to the `SidelobeAttenuation` property.

• **NENBW** — Normalized Effective Noise Bandwidth of the window. This parameter is a measure of the noise performance of the window. It is the width of a rectangular filter that accumulates the same

noise power with the same peak power gain. NENBW can be calculated from the windowing function using the following equation:

$$NENBW = \frac{\sum\limits_{n} w^2(nT)}{\left[\sum\limits_{n} w(nT)\right]^2}$$. The rectangular window has the smallest NENBW, with a value of 1. All other windows have a larger NENBW value. For example, the Hann window has an NENBW value of 1.5.

Measurements Panels
The Measurements panels are the other two panels that appear to the right side of the Spectrum Analyzer figure. These panels are labeled **Trace selection** and **Peak finder**.

Measurements Panel Buttons

Each of the Measurements panels contains the following buttons that enable you to modify the appearance of the current panel.

Button	Description
⊼	Move the current panel to the top. When you are displaying more than one panel, this action moves the current panel above all the other panels.
▼	Collapse the current panel. When you first enable a panel, by default, it displays one or more of its panes. Click this button to hide all of its panes to conserve space. After you click this button, it becomes the expand button ▶.
▶	Expand the current panel. This button appears after you click the collapse button to hide the panes in the current panel. Click this button to display the panes in the current panel and show measurements again. After you click this button, it becomes the collapse button ▼ again.

Button	Description
↗	Undock the current panel. This button lets you move the current panel into a separate window that can be relocated anywhere on your screen. After you click this button, it becomes the dock button ↘ in the new window.
↘	Dock the current panel. This button appears only after you click the undock button. Click this button to put the current panel back into the right side of the Scope window. After you click this button, it becomes the undock button ↗ again.
✕	Close the current panel. This button lets you remove the current panel from the right side of the Scope window.

Some panels have their measurements separated by category into a number of panes. Click the pane expand button ▶ to show each pane that is hidden in the current panel. Click the pane collapse button ▼ to hide each pane that is shown in the current panel.

Trace Selection Panel

When you use the Scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click on any of the other Measurements panels. Choose the signal name for which you would like to display time domain measurements. See the following figure.

You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools > Measurements > Trace Selection**.

Peak Finder Panel

The **Peak Finder** panel displays maxima and the times at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified. You can choose to hide or display the **Peak Finder** panel. In the Spectrum Analyzer menu, select **Tools > Measurements > Peak Finder**. Alternatively, in the Spectrum Analyzer toolbar, select the Peak Finder [icon] button.

The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

Settings Pane

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the MINPEAKHEIGHT parameter, which you can set when you run the findpeaks function.

- **Max Num of Peaks** — The maximum number of peaks to show. This setting is equivalent to the NPEAKS parameter, which you can set when you run the findpeaks function.

- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the MINPEAKDISTANCE parameter, which you can set when you run the findpeaks function.

- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples. Peak excursion is illustrated alongside peak threshold in the following figure.

Whereas the peak threshold is a minimum value necessary for a sample value to be a peak, the peak excursion is the minimum difference between a peak sample and the samples to its left and right in the time domain. In the figure, the green vertical line illustrates the lesser of the two height differences between the labeled peak and its neighboring samples. This height difference must be greater than the **Peak Excursion** value for the labeled peak to be classified as a peak. Compare this setting to peak threshold, which is illustrated by the red horizontal line. The amplitude must be above this horizontal line for the labeled peak to be classified as a peak.

The peak excursion setting is equivalent to the THRESHOLD parameter, which you can set when you run the findpeaks function.

- **Label peaks** — When this check box is selected, the Peak Finder displays the calculated peak values directly on the plot. By default, this check box is cleared and no values are displayed on the plot.

- **Sort by order** — When this check box is selected, the Peak Finder puts the largest calculated peak values in frequency-sorted order. The Peak Finder displays the peak values in the **Peaks** pane. The number of peaks listed is still controlled by the **Max Num of Peaks** parameter. By default, this check box is cleared and the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height.

Peaks Pane

The **Peaks** pane displays all of the largest calculated peak values and the frequencies at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list. You set the order in which the peaks are listed using the **Sort by time** check box.

The numerical values displayed in the **Value** column are equivalent to the pks output argument returned when you run the findpeaks function. The numerical values displayed in the **Frequency** column are similar to the locs output argument returned when you run the findpeaks function.

When you use the zoom options in the Spectrum Analyzer GUI, the peak locations automatically adjust to the frequency range shown in the display. In the Spectrum Analyzer toolbar, click the **Zoom In** or **Zoom X** button to make adjustments. If you constrict the *x*-axis range of the display, then the statistics shown reflect this frequency range. For example, if you zoom in on one peak, the **Peak Finder** panel displays information about only that particular peak.

**Visuals —
Spectrum
Options**

The Visuals:Spectrum Options dialog box controls the visual configuration settings of the Spectrum Analyzer displays. From the Spectrum Analyzer menu, select **View > Properties** to open this dialog box. Alternatively, in the Spectrum Analyzer toolbar, click the properties ⚙ button.

Display Pane

The **Display** pane of the Spectrum Analyzer — Visuals:Spectrum Options dialog box appears as follows.

Title

Specify the display title as a string. By default, the display has no title. Tunable.

Show legend

Select this check box to show the legend in the active display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn the legend off, clear the **Show legend** check box. Tunable.

You can edit the name of any channel in the legend. To do so, double-click the current name, and enter a new channel name. By default, the scope names each channel according to either its signal name or the name of the block from which it comes. If the signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal would have the following default names in the channel legend: `Channel 1`, `Channel 2`.

To change the appearance of any channel of any input signal in the Spectrum Analyzer window, from the Spectrum Analyzer menu, select **View > Style**.

Show grid

When you select this check box, a grid appears in the active display of the Spectrum Analyzer GUI. To hide the grid, clear this check box. Tunable.

Minimum Y-limit

Specify the minimum value of the *y*-axis. Tunable.

Maximum Y-limit

Specify the maximum value of the *y*-axis. Tunable.

Y-axis label

Specify the text for the scope to display to the left of the *y*-axis. Regardless of this property, Spectrum Analyzer always displays power units after this text as one of `'dBm'`, `'dBW'`, `'Watts'`, `'dBm/Hz'`, `'dBW/Hz'`, or `'Watts/Hz'`.Tunable.

Style Dialog Box

In the **Style** dialog box, you can customize the style of displays. You are able to change the color of the figure, the background and foreground colors of the axes, and properties of the lines. From the Spectrum Analyzer menu, select **View > Style** to open this dialog box.

Properties

The **Style** dialog box allows you to modify the following properties of the Spectrum Analyzer GUI:

Figure color
Specify the color that you want to apply to the background of the Spectrum Analyzer GUI. By default, the figure color is gray.

Axes colors
Specify the color that you want to apply to the background of the axes for the display.

Properties for line
Specify the channel for which you want to modify the visibility, line properties, and marker properties.

Visible
Specify whether the selected signal on the display should be visible. If you clear this check box, the line disappears.

Line

Specify the line style, line width, and line color for the selected signal on the display.

Marker

Specify marks for the selected signal on the display to show at data points. This parameter is similar to the Marker property for the MATLAB Handle Graphics plot objects. You can choose any of the marker symbols from the following table.

Specifier	Marker Type
none	No marker (default)
○	Circle
□	Square
×	Cross
●	Point
+	Plus sign
＊	Asterisk
◇	Diamond
▽	Downward-pointing triangle
△	Upward-pointing triangle
◁	Left-pointing triangle
▷	Right-pointing triangle

Specifier	Marker Type
☆	Five-pointed star (pentagram)
✡	Six-pointed star (hexagram)

**Tools —
Plot
Navigation
Options**

The Tools:Plot Navigation Options dialog box allows you to
automatically zoom in on and zoom out of your data. You can also scale
the axes of the Spectrum Analyzer. In the Spectrum Analyzer menu,
select **Tools > Axes Scaling Options** to open this dialog box.

Parameters

The Tools:Plot Navigation Options dialog box appears as follows.

Axis Scaling

Specify when the Scope should automatically scale the axes. You can
select one of the following options:

- Manual — When you select this option, the Scope does not
  automatically scale the axes. You can manually scale the axes in
  any of the following ways:

- ▪ Select **Tools > Scale axes limits**.

- ▪ Press the **Scale Axes Limits** toolbar button.

- ▪ When the scope GUI is the active window, press **Ctrl+A**.

- Auto — When you select this option, the Scope scales the axes as needed, both during and after simulation. Selecting this option enables the **Do not allow Y-axis limits to shrink** check box. By default, this parameter is selected, and the Scope does not shrink the *y*-axis limits when scaling the axes.

- Once at stop — Selecting this option causes the Scope to scale the axes each time simulation is stopped. The Scope does not scale the axes during simulation.

This parameter is Tunable.

Do not allow Y-axis limits to shrink

When you select this parameter, the *y*-axis limits are only allowed to grow during axes scaling operations. If you clear this check box, the *y*-axis limits may shrink during axes scaling operations.

This parameter appears only when you select Auto for the **Axis Scaling** parameter. When you set the **Axis Scaling** parameter to Manual or Once at stop, the *y*-axis limits are allowed to shrink. Tunable.

Y-axis Data range (%)

Set the percentage of the *y*-axis that the Scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to 100, the Scope scales the *y*-axis limits such that your data uses the entire *y*-axis range. If you then set this parameter to 30, the Scope increases the *y*-axis range such that your data uses only 30% of the *y*-axis range. Tunable.

Y-axis Align

Specify where the Scope should align your data with respect to the *y*-axis when it scales the axes. You can select Top, Center, or Bottom. Tunable.

Scale X-axis limits

Check this box to allow the Scope to scale the *x*-axis limits when it scales the axes. Tunable.

X-axis Data range (%)

Set the percentage of the x-axis that the Scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to 100, the Scope scales the x-axis limits such that your data uses the entire x-axis range. If you then set this parameter to 30, the Scope increases the x-axis range such that your data uses only 30% of the x-axis range. Use the x-axis **Align** parameter to specify data placement with respect to the x-axis.

This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

X-axis Align

Specify how the Scope should align your data with respect to the x-axis: Left, Center, or Right. This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

See Also   dsp.TimeScope

Purpose   Create spectrum scope object with same property values

Syntax   `C = clone(H)`

Description   `C = clone(H)` creates a `SpectrumAnalyzer` object `C`, with the same
                 property values as `H`. The `clone` method creates a new unlocked object
                 with uninitialized states.

dsp.SpectrumAnalyzer.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to the
step method. N equals the value of the NumInputPorts property.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

Purpose　　　　Number of outputs of step method

Syntax　　　　N = getNumOutputs(H)

Description　　N = getNumOutputs(H) returns the number of outputs, N, of the step method. The Spectrum Analyzer is a sink object, so N equals 0.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

dsp.SpectrumAnalyzer.hide

Purpose	Hide Spectrum Analyzer window
Syntax	`hide(H)`
Description	`hide(H)` hides the Spectrum Analyzer window, associated with System object, `H`.
See Also	dsp.SpectrumAnalyzer	dsp.SpectrumAnalyzer.show

Purpose   Locked status for input attributes and nontunable properties

Syntax   `isLocked(H)`

Description   `isLocked(H)` returns the locked state of the `SpectrumAnalyzer` object `H`.

The `isLocked` method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the `isLocked` method returns a `true` value.

dsp.SpectrumAnalyzer.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections. This method lets you change any properties or input characteristics.

You should call the `release` method after calling the `step` method when there is no new data for the simulation. When you call the `release` method, the axes will automatically scale in the Spectrum Analyzer figure window. After calling the `release` method, any non-tunable properties can be set once again.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Algorithms   In operation, the `release` method is similar to the `mdlTerminate` function.

See Also   dsp.SpectrumAnalyzer | dsp.SpectrumAnalyzer.reset

Purpose   Reset internal states of Spectrum Analyzer object

Syntax   reset(H)

Description   reset(H) sets the internal states of the SpectrumAnalyzer object H to
                     their initial values.

                     You should call the reset method after calling the step method
                     when you want to clear the Spectrum Analyzer figure displays, prior
                     to releasing system resources. This action enables you to start a
                     simulation from the beginning. When you call the reset method, the
                     displays will become blank again. In this sense, its functionality is
                     similar to that of the MATLAB clf function. Do not call the reset
                     method after calling the release method.

Algorithms   In operation, the reset method is similar to a consecutive execution
                     of the mdlTerminate function and the mdlInitializeConditions
                     function.

See Also   dsp.SpectrumAnalyzer | dsp.SpectrumAnalyzer.release

dsp.SpectrumAnalyzer.show

Purpose   Make Spectrum Analyzer window visible

Syntax   show(H)

Description show(H) makes the Spectrum Analyzer window, associated with System
object, H, visible.

See Also   dsp.SpectrumAnalyzer | dsp.SpectrumAnalyzer.hide

Purpose   Display signal in Spectrum Analyzer figure

Syntax   step(H,X)
                step(H,X1,X2,...,XN)

Description step(H,X) displays the signal, X, in the Spectrum Analyzer figure.

                step(H,X1,X2,...,XN) displays the signals X1, X2,...,XN in the
                Spectrum Analyzer figure when you set the NumInputPorts property
                to N. In this case, X1, X2,...,XN can have different data types and
                dimensions.

dsp.StandardDeviation

Purpose	Standard deviation of input or sequence of inputs

Description

The StandardDeviation object computes the standard deviation for an input or sequence of inputs.

To compute the standard deviation for an input or sequence of inputs:

1 Define and set up your standard deviation object. See "Construction" on page 5-1038.

2 Call step to compute the standard deviation according to the properties of dsp.StandardDeviation. The behavior of step is specific to each object in the toolbox.

Construction

H = dsp.StandardDeviation returns a standard deviation System object, H, that computes the standard deviation for the columns of input.

H = dsp.StandardDeviation('*PropertyName*',*PropertyValue*,...) returns a standard deviation System object, H, with each specified property set to the specified value.

Properties

RunningStandardDeviation

Enable calculation over successive calls to the step method

Set this property to true to enable the calculation of standard deviation over successive calls to the step method. The default is false.

ResetInputPort

Enable resetting in running standard deviation mode

Set this property to true to enable resetting for the running standard deviation. When the property is set to true, you must specify a reset input to the step method to reset the running standard deviation. This property applies only when you set the RunningStandardDeviation property to true. The default is false.

ResetCondition

Reset condition for running standard deviation mode

Specify event to reset the running standard deviation as one of | `Rising edge` | `Falling edge` | `Either edge` | `Non-zero` |. This property applies only when you set the `ResetInputPort` property to `true`. The default is `Non-zero`.

Dimension

Dimension to operate along

Specify how the standard deviation calculation is performed over the data as one of | `All` | `Row` | `Column` | `Custom` |. This property applies only when you set the `RunningStandardDeviation` property to `false`. The default is `Column`.

CustomDimension

Numerical dimension to operate along

Specify the dimension (one-based value) of the input signal, over which the object computes the standard deviation. The cannot exceed the number of dimensions for the input signal. This property applies when you set the `Dimension` property to `Custom`. The default is `1`.

ROIProcessing

ROIProcessing

Enable region of interest processing

Set this property to `true` to enable calculating the standard deviation within a particular region for each image. This property applies only when you set the `RunningStandardDeviation` property to `false` and the `Dimension` property is `All`. The default is `false`.

Full ROI processing support requires a Computer Vision System Toolbox license. With only the DSP System Toolbox license, `Rectangles` is the only selection that applies for the `ROIForm` property.

ROIForm

Type of region of interest

Specify the type of region of interest as one of | Rectangles | Lines | Label matrix | Binary mask |. This property applies only when you set the ROIProcessing property to true. The default is Rectangles.

Full ROI processing support requires a Computer Vision System Toolbox license. With only the DSP System Toolbox license, Rectangles is the only selection that applies for the ROIForm property.

ROIPortion

Calculate over entire ROI or just perimeter

Specify the region over which to calculate the standard deviation as one of | Entire ROI | ROI perimeter |. This property applies if the ROIForm property is Rectangles. The default is Entire ROI.

ROIStatistics

Statistics for each ROI, or one for all ROIs

Specify what statistics to calculate as one of | Individual statistics for each ROI | Single statistic for all ROIs |. This property applies when you set the ROIForm property to Rectangles, Lines, or Label matrix. The default is Individual statistics for each ROI.

ValidityOutputPort

Enable output of validity check of ROI or label numbers

Indicate whether to return the validity of the specified ROI being completely inside image when the ROIForm property is Lines, Rectangles or Binary mask. Indicate whether to return the validity of the specified label numbers when the ROIForm property is Label Matrix. This property applies when you set the ROIForm property to anything except Binary Mask. The default is false.

FrameBasedProcessing

Enable frame-based processing

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. This property applies only when you set the `RunningStandardDeviation` property to `true`. The default is `true`.

Methods

clone	Create standard deviation object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states for running standard deviation computation
step	Calculate standard deviation of input

Examples

Compute running standard deviation of a signal:

```
hstd2 = dsp.StandardDeviation;
hstd2.RunningStandardDeviation = true;
x = randn(100,1);
y = step(hstd2,x);
% y(i) is the standard deviation of all values
% in the vector x(1:i)
```

dsp.StandardDeviation

Algorithms   This object implements the algorithm, inputs, and outputs described
on the Standard Deviation block reference page. The object properties
correspond to the block parameters, except:

- The **Reset port** block parameter corresponds to the ResetInputPort
  and ResetCondition object properties.

- **Treat sample-based row input as a column** block parameter is
  not supported by the dsp.StandardDeviation object.

Both this object and its corresponding block let you specify whether to
process inputs as individual samples or as frames of data. The object
uses the FrameBasedProcessing property. The block uses the **Input
processing** parameter. See "Set the FrameBasedProcessing Property
of a System object" for more information.

See Also   dsp.Variance | dsp.Maximum | dsp.Minimum | dsp.Mean

Purpose   Create standard deviation object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a StandardDeviation object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

dsp.StandardDeviation.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

dsp.StandardDeviation.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the StandardDeviation object
H.

The isLocked method returns a logical value that indicates whether
input attributes and nontunable properties for the object are locked. The
object performs an internal initialization the first time the step method
is executed. This initialization locks nontunable properties and input
specifications, such as dimensions, complexity, and data type of the
input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset states for running standard deviation computation

Syntax   reset(H)

Description   reset(H) sets the internal states of the StandardDeviation object H to their initial values when computing the running standard deviation.

Purpose   Calculate standard deviation of input

Syntax   Y = step(H,X)
                   Y = step(H,X,R)
                   Y = step(H,X,ROI)
                   Y = step(H,X,LABEL,LABELNUMBERS)
                   [Y, FLAG] = step(H,X,ROI)
                   [Y, FLAG] = step(H,X,LABEL,LABELNUMBERS)

Description   Y = step(H,X) computes the standard deviation,Y, of input X. The
                   object computes the standard deviation over successive calls to the step
                   method when the RunningStandardDeviation property is true.

                   Y = step(H,X,R) resets its state based on the value of reset signal R
                   and the ResetCondition property. You can use this option only when
                   the RunningStandardDeviation property is true.

                   Y = step(H,X,ROI) uses additional input ROI as the region of interest
                   when the ROIProcessing property is set to true and the ROIForm
                   property is set to Lines, Rectangles or Binary mask.

                   Y = step(H,X,LABEL,LABELNUMBERS) computes the standard deviation
                   of input image X for region labels contained in vector LABELNUMBERS
                   , with matrix LABEL marking pixels of different regions. This option
                   is available when the ROIProcessing property is set to true and the
                   ROIForm property is set to Label matrix.

                   [Y, FLAG] = step(H,X,ROI) returns FLAG which indicates whether
                   the given region of interest is within the image bounds when the
                   ValidityOutputPort property is true.

                   [Y, FLAG] = step(H,X,LABEL,LABELNUMBERS) returns FLAG which
                   indicates whether the input label numbers are valid when the
                   ValidityOutputPort property is true.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   State-level estimation for bilevel rectangular waveform

Description   The StateLevels object estimates the state levels of a bilevel rectangular waveform.

To estimate the state levels of a bilevel waveform:

1 Define and set up your state-level estimation. See "Construction" on page 5-1051.

2 Call step to estimate the state levels for an input vector according to the properties of dsp.StateLevels. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.StateLevels creates a state-level estimation System object, H, that estimates state levels in a bilevel rectangular waveform using the histogram method with 100 bins.

H = dsp.StateLevels('*PropertyName*',*PropertyValue*,...) returns an StateLevels System object, H, with each specified property set to the specified value.

Properties   **HistogramBounds**

Minimum and maximum levels of the histogram. Specify the range of the histogram as a 2-element real–valued row vector. Signal values outside the range defined by this property are ignored. This property applies when you set the Method property to 'Histogram mode' or 'Histogram mean', and either RunningStateLevels is true, or the HistogramBoundsSource property is set to 'Property'.

Default: [0 5]

HistogramBoundsSource

Source of histogram bounds. Specify how to determine the histogram bounds as one of 'Auto' or 'Property'. When you set this property to 'Auto', the histogram bounds are determined by

the minimum and maximum input values. When you set this property to `'Property'`, the histogram bounds are determined by the value of the `HistogramBounds` property. This property applies when you set the `Method` property to `'Histogram mode'` or `'Histogram mean'`, and the `RunningStateLevels` property is `false`.

Default: `'Auto'`

HistogramNumBins

Number of bins in the histogram. Specify the number of bins in the histogram. This property applies when you set the `Method` property to `'Histogram mode'` or `'Histogram mean'`.

Default: 100

HistogramOutputPort

Enable histogram output. Set this property to `true` to output the histogram used in the computation of the state levels. This property applies when you set the `Method` property to `'Histogram mode'` or `'Histogram mean'`.

Default: false

Method

Algorithm used to compute state levels. Specify the method used to compute state levels as one of `'Histogram mean'`, `'Histogram mode'`, or `'Peak to peak'`.

Default: `'Histogram mode'`

RunningStateLevels

Calculation over successive calls to `step`. Set this property to `true` to enable computation of the state levels over successive calls to the `step`. Otherwise, compute the state levels of the current

input. When you set the `RunningStateLevels` property to `false` and you are using a histogram to compute your state levels, you must set the `HistogramBoundsSource` property to `'Property'`.

Methods

clone	Clones the current instance of the state levels object
getNumInputs	Number of expected inputs to the method
getNumOutputs	Number of outputs of the method
isLocked	Locked status (logical) for input attributes and nontunable properties
plot	Plot signal, state levels, and histogram
release	Allow property value and input characteristics changes
reset	Reset internal states of state levels object
step	Estimate state levels for bilevel rectangular waveform

Definitions

State

A particular level, which can be associated with an upper and lower state boundary. States are ordered from the most negative to the most positive. In a bilevel waveform, the most negative state is the low state. The most positive state is the high state.

State-Level Tolerances

Each state level can have associated lower- and upper-state boundaries. These state boundaries are defined as the state level plus or minus a scalar multiple of the difference between the high state and low state.

To provide a useful tolerance region, the scalar is typically a small number such as 2/100 or 3/100. In general, the α% tolerance region for the low state is defined as

$$S_1 \pm \tfrac{\alpha}{100}(S_2 - S_1)$$

where S_1 is the low-state level and S_2 is the high-state level. Replace the first term in the equation with S_2 to obtain the α% tolerance region for the high state.

The following figure illustrates lower and upper 2% state boundaries (tolerance regions) for a positive-polarity bilevel waveform. The estimated state levels are indicated by a dashed red line.

State Levels ± 2% Tolerance

Examples   **State Levels of 2.3 V Underdamped Noisy Clock**

Compute and plot the state levels of a 2.3 V underdamped noisy clock.

dsp.StateLevels

Load the clock data in the variable, x, and the sampling instants in the variable, t.

```
load('clockex.mat', 'x', 't');
```

Estimate the state levels.

```
 h = dsp.StateLevels;
 levels = step(h, x);
```

Plot the clock data along with the estimated state levels and histograms.

```
plot(h)
```

Algorithms

The StateLevels System object uses the histogram method to estimate the states of a bilevel waveform. The histogram method is described in [1]. To summarize the method:

1 Determine the maximum and minimum amplitudes and amplitude range of the data.

2 For the specified number of histogram bins, determine the bin width as the ratio of the amplitude range to the number of bins.

3 Sort the data values into the histogram bins.

4 Identify the lowest-indexed histogram bin, i_{low}, and highest-indexed histogram bin, i_{high}, with nonzero counts.

5 Divide the histogram into two subhistograms. The lower-histogram bins are $i_{low} \le i \le 1/2(i_{high} - i_{low})$.

The upper-histogram bins are $i_{low} + 1/2(i_{high} - i_{low}) \le i \le i_{high}$.

6 Compute the state levels by determining the mode or mean of the lower and upper histograms.

References

[1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003, pp. 15–17.

See Also   dsp.TransitionMetrics **|** dsp.PulseMetrics **|**

dsp.StateLevels.clone

Purpose   Clones the current instance of the state levels object

Syntax   clone(H)

Description   clone(H) clones the current instance of the state levels object H.

Purpose   Number of expected inputs to the step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to the step method for the state levels object H.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

dsp.StateLevels.getNumOutputs

Purpose   Number of outputs of the `step` method

Syntax   `N = getNumOutputs(H)`

Description   `N = getNumOutputs(H)` returns the number of outputs, N, of the `step` method

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status (logical) for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the state levels object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.StateLevels.plot

Purpose   Plot signal, state levels, and histogram

Syntax   `plot(H)`

Description   `plot(H)` plots the signal and state levels computed in the last call to the `step` method. If the `Method` property of the state levels object is set to `'Histogram mode'` or `'Histogram Mean'`, the histogram is plotted in a subplot below the signal.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.StateLevels.reset

Purpose   Reset internal states of state levels object

Syntax   reset(H)

Description   reset(H) resets the state levels object, H, to clear information
accumulated from previous calls to step.

Purpose   Estimate state levels for bilevel rectangular waveform

Syntax   LEVELS = step(H,X)
                 [LEVELS,HISTOGRAM] = step(H,X)

Description LEVELS = step(H,X) returns a 2-element row vector, LEVELS,
                 containing the estimated state levels for X. X is a real-valued column
                 vector.

                 [LEVELS,HISTOGRAM] = step(H,X) returns a double-precision column
                 vector, HISTOGRAM, containing the histogram of the sample values in
                 X. You can obtain this output only when you set the Method property
                 to either 'Histogram mean' or 'Histogram mode', and you set the
                 HistogramOutputPort property to true.

dsp.SubbandAnalysisFilter

Purpose	Decompose signal into high-frequency and low-frequency subbands

Description

The SubbandAnalysisFilter object decomposes a signal into high-frequency and low-frequency subbands.

To decompose a signal into high-frequency and low-frequency subbands:

1 Define and set up your two-channel subband analysis filter. See "Construction" on page 5-1066.

2 Call step to decompose the signal according to the properties of dsp.SubbandAnalysisFilter. The behavior of step is specific to each object in the toolbox.

Construction

H = dsp.SubbandAnalysisFilter returns a two-channel subband analysis filter, H, that decomposes the input signal into a high-frequency subband and a low-frequency subband, each with half the bandwidth of the input.

H =
dsp.SubbandAnalysisFilter('*PropertyName*',*PropertyValue*,...)
returns a two-channel subband analysis filter, H, with each specified property set to the specified value.

H = dsp.SubbandAnalysisFilter(lpc,hpc,
'*PropertyName*',*PropertyValue*,...) returns a two-channel subband analysis filter, H, with the LowpassCoefficients property set to lpc, the HighpassCoefficients property set to hpc, and other specified properties set to the specified values.

Properties

LowpassCoefficients

Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in descending powers of z. For the lowpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the HighpassCoefficients property. The default values of

this property specify a filter based on a third-order Daubechies wavelet.

HighpassCoefficients

Highpass FIR filter coefficient

Specify a vector of highpass FIR filter coefficients, in descending powers of *z*. For the highpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the `LowpassCoefficients` property. The default values of this property specify a filter based on a third-order Daubechies wavelet.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Action to take when integer input is out of range

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap. This property applies only if the object is not in full precision mode.

CoefficientsDataType

Data type of the coefficients

Specify the FIR filter coefficients fixed-point data type as one of | Same word length as input | Custom |. The default is Same word length as input.

CustomCoefficientsDataType

Coefficients word and fraction lengths

Specify the FIR filter coefficients fixed-point type as a numerictype object with a Signedness of Auto. This property applies only when you set the CoefficientsDataType property to Custom. The default is numerictype([],16,15).

ProductDataType

Data type of product

Specify the product data type as one of | Full precision | Same as input | Custom |. The default is Full precision.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the ProductDataType property to Custom. The default is numerictype([],32,30).

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as one of `Full precision` | `Same as input` | `Same as product` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Data type of output

Specify the output data type as one of | `Same as accumulator` | `Same as product` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,14)`.

Methods

clone	Create subband analysis filter object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties

dsp.SubbandAnalysisFilter

release	Allow property value and input characteristics changes
reset	Reset filter states of subband analysis filter object
step	Decompose signal into high- and low-frequency subbands

Examples

Perfectly reconstruct a signal using subband analysis and synthesis filters:

```
load dspwlets; % load the filter coefficients lod, hid, lor and hir
ha = dsp.SubbandAnalysisFilter(lod, hid);
hs = dsp.SubbandSynthesisFilter(lor, hir);
u = randn(128,1);

[hi, lo] = step(ha, u); % Two channel analysis
y = step(hs, hi, lo); % Two channel synthesis

% Plot difference between original and reconstructed signals
%  with filter latency compensated
plot(u(1:end-7)-y(8:end));
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Two-Channel Analysis Subband Filter block reference page. The object properties correspond to the block parameters, except:

- The SubbandAnalysisFilter object does not have a property that corresponds to the **Input processing** parameter of the Two-Channel Analysis Subband Filter block. The object assumes the input is frame based and always maintains the input frame rate.

- The **Rate options** block parameter is not supported by the dsp.SubbandAnalysisFilter object.

See Also

dsp.SubbandSynthesisFilter | dsp.DyadicAnalysisFilterBank

Purpose　　　Create subband analysis filter object with same property values

Syntax　　　　C = clone(H)

Description　　C = clone(H) create a SubbandAnalysisFilter object C, with the
same property values as H. The clone method creates a new unlocked
object with uninitialized states.

dsp.SubbandAnalysisFilter.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.SubbandAnalysisFilter.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outpust from the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

dsp.SubbandAnalysisFilter.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the SubbandAnalysisFilter object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset filter states of subband analysis filter object

Syntax   reset(H)

Description reset(H) sets the filter states of the SubbandAnalysisFilter object H
to 0.

Purpose	Decompose signal into high- and low-frequency subbands
Syntax	[HI,LO] = step(H,SIG)
Description	[HI,LO] = step(H,SIG) decomposes the input signal, SIG, into a high-frequency subband, HI, and a low-frequency subband, LO.

> **Note** The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

dsp.SubbandSynthesisFilter

Purpose	Reconstruct signal from high-frequency and low-frequency subbands

Description The SubbandSynthesisFilter object reconstructs a signal from high-frequency and low-frequency subbands.

To reconstruct a signal from high-frequency and low-frequency subbands:

1 Define and set up your two-channel subband synthesis filter. See "Construction" on page 5-1078.

2 Call step to reconstruct the signal according to the properties of dsp.SubbandSynthesisFilter. The behavior of step is specific to each object in the toolbox.

Construction H = dsp.SubbandSynthesisFilter returns a two-channel subband synthesis filter, H, that reconstructs a signal from its high-frequency subband and low-frequency subband. Each subband contains half the bandwidth of the original signal.

H = dsp.SubbandSynthesisFilter('*PropertyName*',*PropertyValue*,...) returns a two-channel subband synthesis filter, H, with each specified property set to the specified value.

H = dsp.SubbandSynthesisFilter(lpc,hpc, '*PropertyName*',*PropertyValue*,...) returns a two-channel subband synthesis filter, H. The object has the LowpassCoefficients property set to lpc, the HighpassCoefficients property set to hpc, and other specified properties set to the specified values.

Properties **LowpassCoefficients**

Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in descending powers of z. For the lowpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the HighpassCoefficients property. The default values of

this property specify a filter based on a third-order Daubechies wavelet.

HighpassCoefficients

Highpass FIR filter coefficients

Specify a vector of highpass FIR filter coefficients, in descending powers of *z*. For the highpass filter, use a half-band filter that passes the frequency band stopped by the filter specified in the LowpassCoefficients property. The default values of this property specify a filter based on a third-order Daubechies wavelet.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set FullPrecisionOverride to false, fixed-point data types are controlled through individual fixed-point property settings. For more information, see "Full Precision for Fixed-Point System Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor. This property applies only if the object is not in full precision mode.

OverflowAction

Action to take when integer input is out of range

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap. This property applies only if the object is not in full precision mode.

CoefficientsDataType

Data type of the coefficients

Specify the FIR filter coefficients fixed-point data type as one of | Same word length as input | Custom |. The default is Same word length as input.

CustomCoefficientsDataType

Coefficient word and fraction lengths

Specify the FIR filter coefficients fixed-point type as a numerictype object with a Signedness of Auto. This property applies only when you set the CoefficientsDataType property to Custom. The default is numerictype([],16,15).

ProductDataType

Data type of product

Specify the product data type as one of | Full precision | Same as input | Custom |. The default is Full precision.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies only when you set the ProductDataType property to Custom. The default is numerictype([],32,30).

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as one of | `Full precision` | `Same as input` | `Same as product` | `Custom` |. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Data type of output

Specify the output data type as one of | `Same as accumulator` | `Same as product` | `Same as input` | `Custom` |. The default is `Same as accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,14)`.

Methods

clone	Create subband synthesis filter object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties

dsp.SubbandSynthesisFilter

release	Allow property value and input characteristics changes
reset	Reset internal states of subband synthesis filter object
step	Reconstruct signal from high- and low-frequency subbands

Examples   Perfectly reconstruct a signal using subband analysis and synthesis filters:

```
load dspwlets;  % Load filter coefficients lod,
                % hid, lor and hir
ha = dsp.SubbandAnalysisFilter(lod, hid);
hs = dsp.SubbandSynthesisFilter(lor, hir);
u = randn(128,1);

[hi, lo] = step(ha, u); % Two channel analysis
y = step(hs, hi, lo); % Two channel synthesis

% Plot difference between original and reconstructed
%  signals with filter latency compensated
plot(u(1:end-7)-y(8:end));
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Two-Channel Synthesis Subband Filter block reference page. The object properties correspond to the block parameters, except:

- The SubbandSynthesisFilter object does not have a property that corresponds to the **Input processing** parameter of the Two-Channel Synthesis Subband Filter block. The object only performs sample-based processing and always maintains the input frame rate.

- The **Rate options** block parameter is not supported by the SubbandSynthesisFilter object.

See Also   dsp.SubbandAnalysisFilter | dsp.DyadicSynthesisFilterBank

dsp.SubbandSynthesisFilter.clone

Purpose   Create subband synthesis filter object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a SubbandSynthesisFilter object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

dsp.SubbandSynthesisFilter.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the SubbandSynthesisFilter
object H.

The isLocked method returns a logical value that indicates whether
input attributes and nontunable properties for the object are locked. The
object performs an internal initialization the first time the step method
is executed. This initialization locks nontunable properties and input
specifications, such as dimensions, complexity, and data type of the
input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset internal states of subband synthesis filter object

Syntax   reset(H)

Description   reset(H) sets the internal states of the SubbandSynthesisFilter
                   object H to their initial values.

Purpose	Reconstruct signal from high- and low-frequency subbands
Syntax	Y = step(H,HI,LO)
Description	Y = step(H,HI,LO) reconstructs a signal from a high-frequency subband, HI, and a low-frequency subband, LO.

> **Note** The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose   Time domain signal display

Description   The TimeScope object displays time-domain signals.

To display time-domain signals in the Time Scope:

1 Define and set up your Time Scope. See "Construction" on page 5-1093.

2 Call step to display the signal in the Time Scope figure. The behavior of step is specific to each object in the toolbox.

Use the MATLAB clear function to close the Time Scope figure window.

See the following sections for information on the Time Scope System object:

- "Construction" on page 5-1093

See the following sections for information on the Time Scope Graphical User Interface:

For examples on how to use the Time Scope System object, see "Examples" on page 5-1153.

Note For information about the Time Scope block, see Time Scope.

Construction   H = dsp.TimeScope returns a Time Scope System object, H. This object displays real- and complex-valued floating and fixed-point signals in the time domain.

H = dsp.TimeScope('*PropertyName*',*PropertyValue*,...) creates a Time Scope System object, H, with each specified property *PropertyName* set to the specified value.

```
H =
dsp.TimeScope(NUMINPUTS,SAMPLERATE,'PropertyName',PropertyValue,...)
```

creates a Time Scope System object, `H`. This object has the `NumInputPorts` property set to *NUMINPUTS*, the `SampleRate` property set to *SAMPLERATE*, and other specified properties set to the specified values. *NUMINPUTS* and *SAMPLERATE* are value-only arguments. You can specify *PropertyName–PropertyValue* arguments in any order.

Properties

ActiveDisplay

Active display for display-specific properties

Specify as an integer the active display number for getting and setting relevant properties. The number of a display corresponds to its column-wise placement index. Setting this property controls which display is used for `YLimits`, `YLabel`, `LegendSource`, `Grid`, `Title`, and `MagnitudePhase`. This property is Tunable.

Default: 1

FrameBasedProcessing

Process input in frames or as samples

When you set this property to `true`, you enable frame-based processing. When you set this property to `false`, you enable sample-based processing. This property is Tunable.

Default: `true`

Grid

Option to enable or disable grid display

When you set this property to `true`, the grid appears. When you set this property to `false`, the grid does not appear. When set, `ActiveDisplay` controls which display is updated. This property is Tunable.

Default: `false`

LayoutDimensions

Layout grid dimensions

Specify the layout grid dimensions as a 2-element vector:
`[numberOfRows, numberOfColumns]`. You may use no more than
four rows or four columns. This property is Tunable.

Default: `[1,1]`

LegendSource

Source of legend

Specify the source of the legend as either `None` or `Auto`. When
you set this property to `'None'`, the Time Scope does not display
a legend. When you set this property to `'Auto'`, the Time Scope
displays a legend with automatic string labels for each input
channel. See FrameBasedProcessing for information on input
channels. This property is Tunable.

Default: `'None'`

MagnitudePhase

Plot signal as Magnitude-and-Phase

Set this property to true to plot the magnitude and phase of the
input signal on two separate axes within the same active display.
When set, `ActiveDisplay` controls which displays are updated.
The active display shows the magnitude of the input signal on the
top axes and its phase, in degrees, on the bottom axes.

This property is particularly useful for complex-valued input
signals. When the input signal is real-valued and you select this
check box, the phase will be 0 degrees when the amplitude of
the input signal is nonnegative and 180 degrees when the input
signal is negative. This property is Tunable.

Default: `false`

MaximizeAxes

Maximize axes control

Specify whether to display the Time Scope in maximized axes mode. In this mode, each of the axes is expanded to fit into the entire display. To conserve space, labels do not appear in each display. Instead, tick-mark values appear on top of the plotted data. You can select one of the following options:

- `Auto` — In this mode, the axes appear maximized in all displays only if the `Title` and `YLabel` properties are empty for every display. If you enter any value in any display for either of these properties, the axes are not maximized.

- `On` — In this mode, the axes appear maximized in all displays. Any values entered into the `Title` and `YLabel` properties are hidden.

- `Off` — In this mode, none of the axes appear maximized.

This property is Tunable.

Default: `'Auto'`

Name

Caption to display on Time Scope window

Specify the caption to display on the scope window as any string. This property is Tunable.

Default: `'Time Scope'`

NumInputPorts

Number of input signals

Specify the number of input signals to display on the Time Scope as a positive integer. You must invoke the `step` method with the same number of inputs as the value of this property.

Default: 1

PlotType

Option to control the type of plot

Specify the type of plot to use for all the input signals displayed in the Time Scope GUI.

- When you set this property to `'Line'`, Time Scope displays the input signal as lines connecting each of the sampled values. This approach is similar to the functionality of the MATLAB `line` or `plot` function.

- When you set this property to `'Stairs'`, Time Scope displays the input signal as a *stairstep* graph. A stairstep graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. This approach is equivalent to the MATLAB `stairs` function. Stairstep graphs are useful for drawing time history graphs of digitally sampled data.

This property is Tunable.

Default: `'Line'`

Position

Time Scope window position in pixels

Specify, in pixels, the size and location of the Time Scope window as a 4-element double vector of the form, `[left bottom width height]`. You can place the Time Scope window in a specific position on your screen by modifying the values to this property. This property is Tunable.

Default: The default depends on your screen resolution. By default, the Time Scope window is placed in the center of your screen with a width of 410 pixels and height of 300 pixels.

ReduceUpdates

Reduce updates to improve performance

When you set this property to true, the Time Scope logs data for later use and updates the window periodically. When you set this property to false, the Time Scope updates every time the step method is called. The simulation speed is faster when this property is set to true. This property is Tunable.

You can also modify this property from the Time Scope GUI. Opening the **Simulation** menu and clearing the **Reduce Updates to Improve Performance** check box is the same as setting this property to false.

Default: true

SampleRate

Sample rate of inputs

Specify the sampling rate, in hertz, of any input signal, as a scalar or as a numeric vector with length equal to the value of NumInputPorts. The inverse of the sample rate determines the spacing between points on the *x*-axis (time axis) in the displayed signal. When you set SampleRate to a scalar value and NumInputPorts is greater than 1, the object uses the same sample rate for all inputs.

Default: 1

TimeAxisLabels

Time-axis labels

Specify how time-axis labels should appear in the Time Scope displays as one of `'All'`, `'Bottom`, or `'None'`.

- When you set this property to `'All'`, time-axis labels appear in all displays.

- When you set this property to `'Bottom'`, time-axis labels appear in the bottom display of each column.

- When you set this property to `'None'`, there are no labels in any displays.

This property is Tunable.

Default: `'All'`

TimeDisplayOffset

Time display offset

Specify the offset, in seconds, to apply to the *x*-axis (time axis). This property can be either a numeric scalar or a vector of length equal to the number of input channels. If you specify this property as a scalar, then that value is the time display offset for all channels. If you specify a vector, each vector element is the time offset for the corresponding channel. For vectors with length less than the number of input channels, the time display offsets for the remaining channels are set to 0. If a vector has a length greater than the number of input channels, the extra vector elements are ignored. This property is Tunable.

See FrameBasedProcessing for information on input channels. See `TimeSpan` and `TimeSpanSource` for information on the *x*-axis limits and time span settings.

Default: 0

TimeSpan

Specify the time span, in seconds, as a positive, numeric scalar value. This property applies when FrameBasedProcessing is false. This property also applies when FrameBasedProcessing is true and TimeSpanSource is Property. The *x*-axis limits are calculated as follows.

Minimum *x*-axis limit = min(TimeDisplayOffset)
Maximum *x*-axis limit = max(TimeDisplayOffset) + TimeSpan

where TimeDisplayOffset and TimeSpan are the values of their respective properties. This property is Tunable.

Default: 10

TimeSpanOverrunMode

Wrap or scroll when the TimeSpan value is overrun

Specify how the Time Scope displays new data beyond the visible time span as one of 'Wrap' or 'Scroll'.

- When you set this property to Wrap, the Time Scope displays new data up until it reaches the maximum *x*-axis limit. When the data reaches the maximum *x*-axis limit of the Time Scope window, the Time Scope clears the display. The Time Scope then updates the time offset value and begins displaying subsequent data points starting from the minimum *x*-axis limit.

- When you set this property to Scroll, the Time Scope scrolls old data to the left to make room for new data on the right side of the Time Scope display. This mode is graphically intensive and can affect run-time performance, but it is beneficial for debugging and for monitoring time-varying signals.

This property is Tunable.

Default: 'Wrap'

TimeSpanSource

Source of time span

Specify the source of the time span for frame-based input signals as one of 'Auto' or 'Property'. This property applies when FrameBasedProcessing is set to true. When you set this property to 'Property', the object derives the *x*-axis limits from the TimeDisplayOffset and TimeSpan properties. When you set this property to Auto, the *x*-axis limits are the following:

Minimum *x*-axis limit = min(TimeDisplayOffset)
Maximum *x*-axis limit = max(TimeDisplayOffset) + max(1/SampleRate.**FrameSize*)

where TimeDisplayOffset and SampleRate are the values of their respective properties. *FrameSize* is a vector equal to the number of rows in each input signal.

This property is Tunable.

Default: 'Property'

TimeUnits

Units of the time axis

Specify the units used to describe the time axis.

- When you set this property to 'Metric', the Time Scope converts the time axis values to the most appropriate metric units, such as microseconds, milliseconds, minutes, days, etc.

- When you set this property to 'Seconds', the Time Scope will always leave the time units as seconds.

- When you set this property to 'None', the Time Scope will not display any units on the time axis.

This property is Tunable.

Default: 'Metric'

Title

Display title

Specify the display title as a string. When set, `ActiveDisplay` controls which display is updated. This property is Tunable.

Default: `''`

YLabel

The label for the *y*-axis

Specify the *y*-axis label as a string. This property applies only when MagnitudePhase is `false`. When MagnitudePhase is `true`, the two *y*-axis labels are read-only values that are set to `'Magnitude'` and `'Phase'` for the magnitude plot and the phase plot, respectively. When set, `ActiveDisplay` controls which display is updated. This property is Tunable.

Default: `'Amplitude'` if MagnitudePhase is `false`

YLimits

The limits for the *y*-axis

Specify the *y*-axis limits as a 2-element numeric vector, `[ymin ymax]`. When set, `ActiveDisplay` controls which display is updated.

When MagnitudePhase is `true`, this property specifies the *y*-axis limits of only the magnitude plot. The *y*-axis limits of the phase plot are always `[-180, 180]`. This property is Tunable.

Default: `[-10, 10]` if MagnitudePhase is `false`, `[0, 10]` if MagnitudePhase is `true`.

Methods	clone	Create time scope object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	hide	Hide time scope window
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	reset	Reset internal states of time scope object
	show	Make time scope window visible
	step	Display signal in time scope figure

Displaying Multiple Signals

Multiple Signal Input

You can configure the Time Scope to show multiple signals within the same display or on separate displays. By default, the signals appear as different-colored lines on the same display. The signals can have different dimensions, sample rates, and data types. Each signal can be either real or complex valued. You can set the number of input ports on the Time Scope in the following ways:

- Set the NumInputPorts property. This property is nontunable, so you should set it before you run the step method.

- Run the show method to open the Scope window. In the Scope menu, select **File > Number of Input Ports**.

An input signal may contain multiple channels, depending on its dimensions. Multiple channels of data always appear as different-colored lines on the same display.

Multiple Displays

You can display multiple channels of data on different displays in the Scope window. In the Scope toolbar, select **View > Layout**, or select

the Layout button (⊞). This feature allows you to tile the window into a number of separate displays, up to a grid of 4 rows and 4 columns. For example, if there are three inputs to the Scope, you can display the signals in separate displays by selecting row 3, column 1, as shown in the following figure.

After you select row 3, column 1, the Scope window has three separate displays, as shown in the following figure.

Signal Display

Time Scope uses the **Time span** and **Time display offset** parameters in order to determine the time range. To change the signal display settings, select **View > Properties** to bring up the Visuals:Time Domain Options dialog box. Then, modify the values for the **Time span** and **Time display offset** parameters on the **Main** tab. For example, if you set the **Time span** to 25 seconds, the scope displays 25 seconds' worth of simulation data at a time. If you also set the **Time display offset** to 5 seconds, the scope displays values on the time axis from 5 to 30 seconds. The values on the *time*-axis of the Time Scope display remain the same throughout simulation.

To communicate the simulation time that corresponds to the current display, the scope uses the **Time units**, **Time offset**, and **Simulation time** indicators on the scope window. The following figure highlights these and other important aspects of the Time Scope window.

Time Indicators

- **Minimum time-axis limit** — The Time Scope sets the minimum *time*-axis limit using the value of the **Time display offset** parameter on the **Main** tab of the Visuals:Time Domain Options dialog box. If you specify a vector of values for the **Time display offset** parameter, the scope uses the smallest of those values to set the minimum *time*-axis limit.

- **Maximum time-axis limit** — The Time Scope sets the maximum *time*-axis limit by summing the value of **Time display offset** parameter with the value of the **Time span** parameter. If you specify a vector of values for the **Time display offset** parameter, the scope sets the maximum *time*-axis limit by summing the largest of those values with the value of the **Time span** parameter.

- **Time units** — The units used to describe the *time*-axis. The Time Scope sets the time units using the value of the **Time units** parameter on the **Main** tab of the Visuals:Time Domain Options dialog box. By default, this parameter is set to Metric (based on Time Span) and displays in metric units such as milliseconds, microseconds, minutes, days, etc. You can change it to Seconds to always display the *time*-axis values in units of seconds. You can change it to None to not display any units on the *time*-axis. When you set this parameter to None, then Time Scope shows only the word Time on the *time*-axis.

To hide both the word Time and the values on the *time*-axis, set
the **Show time-axis labels** parameter to None. To hide both the
word Time and the values on the *time*-axis in all displays except the
bottom ones in each column of displays, set this parameter to Bottom
Displays Only. This behavior differs from the Simulink Scope block,
which always shows the values but never shows a label on the *x*-axis.

For more information, see "Visuals — Time Domain Options" on page
5-1140.

Simulation Indicators

- **Simulation status** — Provides the current status of the model
  simulation. The status can be one of the following conditions:

  - Processing — Occurs after you run the step method and before
    you run the release method.

  - Stopped — Occurs after you construct the TimeScope object and
    before you first run the step method. This status also occurs after
    you run the release method.

  The **Simulation status** is part of the **Status Bar** in the Time Scope
  window. You can choose to hide or display the entire **Status Bar**.
  From the Time Scope menu, select **View > Status Bar**.

- **Time offset** — The **Time offset** value helps you determine the
  simulation times for which the scope is displaying data. The value is
  always in the range 0≤ Time offset≤ **Simulation time**.

  Suppose you set the **Time span** to 20 seconds, and you see a **Time
  offset** of 0 (secs) on the scope window. This value indicates that
  the scope is displaying data for the first 0 to 20 seconds of simulation
  time. If the **Time offset** changes to 20 (secs), the scope displays
  data for simulation times from 20 seconds to 40 seconds. The scope
  continues to update the **Time offset** value until the simulation is
  complete.

- **Simulation time** — The amount of time that has progressed since
  the Time Scope object was first constructed or reset. Every time you

call the step method, the simulation time increases by the number of rows in the input signal divided by the sample rate, as given by

the following formula: $t_{sim} = t_{sim} + \dfrac{\text{length}(x)}{SampleRate}$. You can modify the **SampleRate** parameter on the **Main options** pane of the **Spectrum Settings** panel. You can also set the sample rate using the SampleRate property.

The **Simulation time** is part of the **Status Bar** in the Time Scope window. You can choose to hide or display the entire **Status Bar**. From the Time Scope menu, select **View > Status Bar** .

Axes Maximization

You can specify whether to display the Time Scope in maximized axes mode. In this mode, the axes are expanded to fill the entire display. In each display, there is no space to show titles or axis labels. The minimum and maximum *time*-axis limits are located at the far-left and far-right edges of the display. The values at the axis tick marks appear as grid lines on top of the axes. The following figure highlights the important indicators in this mode of the Time Scope window.

To enable this mode, in the Time Scope menu, select **View > Properties** to bring up the Visuals:Time Domain Options dialog box. In the **Main** pane, you can set the **Maximize axes** parameter to one of the following options:

- Auto — In this mode, the axes appear maximized in all displays only if the **Title** and **Y-Axis label** parameters are empty for every display. If you enter any value in any display for either of these parameters, the axes are not maximized.

- On — In this mode, the axes appear maximized in all displays. Any values entered into the **Title** and **Y-Axis label** parameters are hidden.

- Off — In this mode, none of the axes appear maximized.

You can also set the MaximizeAxes property to one of the aforementioned options.

Reduce Updates to Improve Performance

By default, the Time Scope updates the displays periodically at a rate not exceeding 20 Hz. If you would like the Time Scope to update on every simulation time step, you can disable the **Reduce Updates to Improve Performance** option. In the Time Scope menu, select **Playback > Reduce Updates to Improve Performance** to clear the check box. You can also set the ReduceUpdates property to false to disable this option. However, as a recommended practice, you can leave this option enabled because doing so can significantly improve the speed of the simulation.

Toolbar   The Time Scope toolbar contains the following buttons.

Axes Control Buttons

	Tools > Zoom In	N/A	When this tool is active, you can zoom in on the scope window. To do so, click in the center of your area of interest, or click and drag your cursor to draw a rectangular area of interest inside the scope window.
	Tools > Zoom X	N/A	When this tool is active, you can zoom in on the x-axis. To do so, click inside the scope window, or click and drag your cursor along the x-axis over your area of interest.
	Tools > Zoom Y	N/A	When this tool is active, you can zoom in on the y-axis. To do so, click inside the scope window, or click and drag your cursor along the y-axis over your area of interest.
	Tools > Scale Axes Limits	Ctrl+A	Click this button to scale the axes in the active scope window. Alternatively, you can enable automatic axes scaling by selecting one of the following options from the **Tools** menu: • **Automatically Scale Axes Limits** — When you select this option, the scope scales the axes as needed during simulation. • **Scale Axes Limits at Stop** — When you select this option, the scope scales the axes each time the simulation is stopped.

Measurements Buttons

⊞	**Tools > Measurements > Cursor Measurements**	N/A	Open or close the **Cursor Measurements** panel. This panel puts screen cursors on all the displays. See the "Cursor Measurements Panel" on page 5-1114 section for more information.
◠	**Tools > Measurements > Signal Statistics**	N/A	Open or close the **Signal Statistics** panel. This panel displays the maximum, minimum, peak-to-peak difference, mean, median, RMS values of a selected signal, and the times at which the maximum and minimum occur. See the "Signal Statistics Panel" on page 5-1116 section for more information.
⊓	**Tools > Measurements > Bilevel Measurements**	N/A	Open or close the **Bilevel Measurements** panel. This panel displays information about a selected signal's transitions, overshoots or undershoots, and cycles. See the "Bilevel Measurements Panel" on page 5-1118 section for more information.
⏶	**Tools > Measurements > Peak Finder**	N/A	Open or close the **Peak Finder** panel. This panel displays maxima and the times at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified.

			See the "Peak Finder Panel" on page 5-1132 section for more information.

Other Buttons

⊞	**View > Layout**	N/A	Arrange the layout of displays in the Time Scope. This feature allows you to tile your screen into a number of separate displays, up to a grid of 4 rows and 4 columns. You may find multiple displays useful when the Time Scope takes multiple input signals. The default display is 1 row and 1 column. See the "Multiple Displays" on page 5-1104 section for more information.

You can control whether this toolbar appears in the Time Scope window. From the Time Scope menu, select **View > Toolbar**.

Measurements Panels The Measurements panels are the five panels that appear to the right side of the Time Scope GUI. These panels are labeled **Trace selection**, **Cursor measurements**, **Signal statistics**, **Bilevel measurements**, and **Peak finder**.

The Time Domain Measurements panels only appear if the **Measurements** tool is enabled in the Time Scope — Tools:Plot Navigation Options dialog box. To open this dialog box, in the Time Scope menu, select **File > Configuration** and click the **Tools** pane. If you disable the tool by clearing the **Enabled** check box, the Time Domain Measurements tools no longer display in the Time Scope GUI. You can reenable the tool at any time by selecting the **Enabled** check box. See the "Tools — Plot Navigation Options" on page 5-1149 section for more information.

Measurements Panel Buttons

Each of the Measurements panels contains the following buttons that enable you to modify the appearance of the current panel.

Button	Description
	Move the current panel to the top. When you are displaying more than one panel, this action moves the current panel above all the other panels.
	Collapse the current panel. When you first enable a panel, by default, it displays one or more of its panes. Click this button to hide all of its panes to conserve space. After you click this button, it becomes the expand button.
	Expand the current panel. This button appears after you click the collapse button to hide the panes in the current panel. Click this button to display the panes in the current panel and show measurements again. After you click this button, it becomes the collapse button again.
	Undock the current panel. This button lets you move the current panel into a separate window that can be relocated anywhere on your screen. After you click this button, it becomes the dock button in the new window.
	Dock the current panel. This button appears only after you click the undock button. Click this button to put the current panel back into the right side of the Scope window. After you click this button, it becomes the undock button again.
	Close the current panel. This button lets you remove the current panel from the right side of the Scope window.

Some panels have their measurements separated by category into a number of panes. Click the pane expand button ► to show each pane that is hidden in the current panel. Click the pane collapse button ▼ to hide each pane that is shown in the current panel.

Trace Selection Panel

When you use the Scope to view multiple signals, the Trace Selection panel appears if you have more than one signal displayed and you click on any of the other Measurements panels. Choose the signal name for which you would like to display time domain measurements. See the following figure.

You can choose to hide or display the **Trace Selection** panel. In the Scope menu, select **Tools > Measurements > Trace Selection**.

Cursor Measurements Panel

The **Cursor Measurements** panel displays screen cursors. You can choose to hide or display the **Cursor Measurements** panel. In the Scope menu, select **Tools > Measurements > Cursor Measurements**. Alternatively, in the Scope toolbar, click the Cursor Measurements button.

Settings Pane

The **Settings** pane enables you to modify the type of screen cursors used to calculate time and value measurements.

- **Screen Cursors**— Shows screen cursors.

- **Horizontal**— Shows horizontal screen cursors.

- **Vertical**— Shows vertical screen cursors.

- **Waveform Cursors**— Shows cursors that attach to the input signals.

- **Lock Cursor Spacing**— Locks the time difference between the two cursors.

Measurements Pane

The **Measurements** pane shows the time and value measurements.

- **1 |**— Shows or enables you to modify the time or value at cursor number one, or both.

- **2 :**— Shows or enables you to modify the time or value at cursor number two, or both.

- **Δt**— Shows the absolute value of the difference in the times between cursor number one and cursor number two.

- **ΔV**— Shows the absolute value of the difference in signal amplitudes between cursor number one and cursor number two.

- **1/Δt**— Shows the rate, the reciprocal of the absolute value of the difference in the times between cursor number one and cursor number two.

- **ΔV/Δt**— Shows the scope, the ratio of the absolute value of the difference in signal amplitudes between cursors to the absolute value of the difference in the times between cursors.

Signal Statistics Panel

The **Signal Statistics** panel displays the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal. It also shows the times at which the maximum and minimum values occur. You can choose to hide or display the **Signal Statistics** panel. In the Scope menu, select **Tools > Measurements > Signal Statistics**. Alternatively, in the Scope toolbar, click the Signal

Statistics ![icon] button.

⊼ ▼ Signal Statistics		↗ ✕
	Value	Time (us)
Max	2.343	22.500
Min	-57.337 m	5.000
Peak to Peak	2.400	
Mean	1.150	
Median	1.152	
RMS	1.628	

Signal Statistics Measurements

The **Signal Statistics** panel shows statistics about the portion of the input signal within the *x*-axis and *y*-axis limits of the active display. The statistics shown are:

- **Max** — Shows the maximum or largest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `max` function reference.

- **Min** — Shows the minimum or smallest value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `min` function reference.

- **Peak to Peak** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `peak2peak` function reference.

- **Mean** — Shows the average or mean of all the values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `mean` function reference.

- **Median** — Shows the median value within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the MATLAB `median` function reference.

- **RMS** — Shows the difference between the maximum and minimum values within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `rms` function reference.

When you use the zoom options in the Scope, the Signal Statistics measurements automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the *x*-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one pulse to make the **Signal Statistics** panel display information about only that particular pulse.

The Signal Statistics measurements are valid for any units of the input signal. The letter after the value associated with each measurement represents the appropriate International System of Units (SI) prefix, such as *m* for *milli-*. For example, if the input signal is measured in volts, an *m* next to a measurement value indicates that this value is in units of millivolts.

Bilevel Measurements Panel

The **Bilevel Measurements** panel shows information about a selected signal's transitions, overshoots or undershoots, and cycles. You can choose to hide or display the **Bilevel Measurements** panel. In the Scope menu, select **Tools > Measurements > Bilevel Measurements**. Alternatively, in the Scope toolbar, you can select the Bilevel Measurements button.

The **Bilevel Measurements** panel is separated into four panes, labeled **Settings**, **Transitions**, **Overshoots / Undershoots**, and **Cycles**. You can expand each pane to see the available options.

Settings Pane

The **Settings** pane enables you to modify the properties used to calculate various measurements involving transitions, overshoots, undershoots, and cycles. You can modify the high-state level, low-state level, state-level tolerance, upper-reference level, mid-reference level, and lower-reference level, as shown in the following figure.

- **Auto State Level** — When this check box is selected, the Bilevel measurements panel autodetects the high- and low- state levels of a bilevel waveform. For more information on the algorithm this option uses, see the Signal Processing Toolbox `statelevels` function

reference. When this check box is cleared, you may enter in values
for the high- and low- state levels manually.

- **High** — Used to manually specify the value that denotes a positive
  polarity, or high-state level, as shown in the following figure.

Positive Polarity Pulse

- **Low** — Used to manually specify the value that denotes a negative
  polarity, or low-state level, as shown in the following figure.

Negative Polarity Pulse

- **State Level Tolerance** — Tolerance within which the initial and final levels of each transition must be within their respective state levels. This value is expressed as a percentage of the difference between the high- and low-state levels.

- **Upper Ref Level** — Used to compute the end of the rise-time measurement or the start of the fall time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.

- **Mid Ref Level** — Used to determine when a transition occurs. This value is expressed as a percentage of the difference between the high- and low- state levels. In the following figure, the mid-reference level is shown as the horizontal line, and its corresponding mid-reference level instant is shown as the vertical line.

- **Lower Ref Level** — Used to compute the end of the fall-time measurement or the start of the rise-time measurement. This value is expressed as a percentage of the difference between the high- and low-state levels.

- **Settle Seek** — The duration after the mid-reference level instant when each transition occurs used for computing a valid settling time. This value is equivalent to the input parameter, D, which you can set when you run the settlingtime function. The settling time is displayed in the **Overshoots/Undershoots** pane.

Transitions Pane

The **Transitions** pane displays calculated measurements associated with the input signal changing between its two possible state level values, high and low.

A positive-going transition, or *rising edge*, in a bilevel waveform is a transition from the low-state level to the high-state level. A positive-going transition has a slope value greater than zero. The following figure shows a positive-going transition.

X(t)

$$\frac{dX}{dt} > 0$$

Whenever there is a plus sign (+) next to a text label, this symbol refers to measurement associated with a rising edge, a transition from a low-state level to a high-state level.

A negative-going transition, or falling edge, in a bilevel waveform is a transition from the high-state level to the low-state level. A negative-going transition has a slope value less than zero. The following figure shows a negative-going transition.

Whenever there is a minus sign (–) next to a text label, this symbol refers to measurement associated with a falling edge, a transition from a high-state level to a low-state level.

The Transition measurements assume that the amplitude of the input signal is in units of volts. You must convert all input signals to volts for the Transition measurements to be valid.

- **High** — The high-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals:Time Domain Options dialog box. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.

- **Low** — The low-amplitude state level of the input signal over the duration of the **Time Span** parameter. You can set **Time Span** in the **Main** pane of the Visuals:Time Domain Options dialog box. For

more information on the algorithm this measurement uses, see the Signal Processing Toolbox `statelevels` function reference.

- **Amplitude** — Difference in amplitude between the high-state level and the low-state level.

- **+ Edges** — Total number of positive-polarity, or rising, edges counted within the displayed portion of the input signal.

- **+ Rise Time** — Average amount of time required for each rising edge to cross from the lower-reference level to the upper-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `risetime` function reference.

- **+ Slew Rate** — Average slope of each rising-edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. The region in which the slew rate is calculated appears in gray in the following figure.

For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.

- **– Edges** — Total number of negative-polarity or falling edges counted within the displayed portion of the input signal.

- **– Fall Time** — Average amount of time required for each falling edge to cross from the upper-reference level to the lower-reference level. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `falltime` function reference.

- **– Slew Rate** — Average slope of each falling edge transition line within the upper- and lower-percent reference levels in the displayed portion of the input signal. For more information on the algorithm

this measurement uses, see the Signal Processing Toolbox `slewrate` function reference.

Overshoots/Undershoots

The **Overshoots/Undershoots** pane displays calculated measurements involving the distortion and damping of the input signal. *Overshoot* and *undershoot* refer to the amount that a signal respectively exceeds and falls below its final steady-state value. *Preshoot* refers to the amount prior to a transition that a signal varies from its initial steady-state value. This figure shows preshoot, overshoot, and undershoot for a rising-edge transition.

The next figure shows preshoot, overshoot, and undershoot for a falling-edge transition.

- **+ Preshoot** — Average lowest aberration in the region immediately preceding each rising transition.

- **+ Overshoot** — Average highest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.

- **+ Undershoot** — Average lowest aberration in the region immediately following each rising transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.

- **+ Settling Time** — Average time required for each rising edge to enter and remain within the tolerance of the high-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the high-state level. This crossing is illustrated in the following figure.

You can modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

- – **Preshoot** — Average highest aberration in the region immediately preceding each falling transition.

- – **Overshoot** — Average highest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `overshoot` function reference.

- – **Undershoot** — Average lowest aberration in the region immediately following each falling transition. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `undershoot` function reference.

- – **Settling Time** — Average time required for each falling edge to enter and remain within the tolerance of the low-state level for the remainder of the settle seek duration. The settling time is the time after the mid-reference level instant when the signal crosses into and remains in the tolerance region around the low-state level. You can

modify the settle seek duration parameter in the **Settings** pane. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `settlingtime` function reference.

Cycles

The **Cycles** pane displays calculated measurements pertaining to repetitions or trends in the displayed portion of the input signal.

- **Period** — Average duration between adjacent edges of identical polarity within the displayed portion of the input signal. The Bilevel measurements panel calculates period as follows. It takes the difference between the mid-reference level instants of the initial transition of each positive-polarity pulse and the next positive-going transition. These mid-reference level instants appear as red dots in the following figure.

For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulseperiod` function reference.

- **Frequency** — Reciprocal of the average period. Whereas period is typically measured in some metric form of seconds, or seconds per cycle, frequency is typically measured in hertz or cycles per second.

- **+ Pulses** — Number of positive-polarity pulses counted.

- **+ Width** — Average duration between rising and falling edges of each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.

- **+ Duty Cycle** — Average ratio of pulse width to pulse period for each positive-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.

- **– Pulses** — Number of negative-polarity pulses counted.

- **– Width** — Average duration between rising and falling edges of each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `pulsewidth` function reference.

- **– Duty Cycle** — Average ratio of pulse width to pulse period for each negative-polarity pulse within the displayed portion of the input signal. For more information on the algorithm this measurement uses, see the Signal Processing Toolbox `dutycycle` function reference.

When you use the zoom options in the Scope, the bilevel measurements automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the x-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one rising edge to make the **Bilevel Measurements** panel display information about only that

particular rising edge. However, this feature does not apply to the **High** and **Low** measurements.

Peak Finder Panel

The **Peak Finder** panel displays maxima and the times at which they occur, allowing the settings for peak threshold, maximum number of peaks, and peak excursion to be modified. You can choose to hide or display the **Peak Finder** panel. In the Scope menu, select **Tools > Measurements > Peak Finder**. Alternatively, in the Scope toolbar, select the Peak Finder button.

The **Peak finder** panel is separated into two panes, labeled **Settings** and **Peaks**. You can expand each pane to see the available options.

Settings Pane

The **Settings** pane enables you to modify the parameters used to calculate the peak values within the displayed portion of the input

signal. For more information on the algorithms this pane uses, see the Signal Processing Toolbox `findpeaks` function reference.

- **Peak Threshold** — The level above which peaks are detected. This setting is equivalent to the `MINPEAKHEIGHT` parameter, which you can set when you run the `findpeaks` function.

- **Max Num of Peaks** — The maximum number of peaks to show. This setting is equivalent to the `NPEAKS` parameter, which you can set when you run the `findpeaks` function.

- **Min Peaks Distance** — The minimum number of samples between adjacent peaks. This setting is equivalent to the `MINPEAKDISTANCE` parameter, which you can set when you run the `findpeaks` function.

- **Peak Excursion** — The minimum height difference between a peak and its neighboring samples. Peak excursion is illustrated alongside peak threshold in the following figure.

Whereas the peak threshold is a minimum value necessary for a sample value to be a peak, the peak excursion is the minimum difference between a peak sample and the samples to its left and right

in the time domain. In the figure, the green vertical line illustrates the lesser of the two height differences between the labeled peak and its neighboring samples. This height difference must be greater than the **Peak Excursion** value for the labeled peak to be classified as a peak. Compare this setting to peak threshold, which is illustrated by the red horizontal line. The amplitude must be above this horizontal line for the labeled peak to be classified as a peak.

The peak excursion setting is equivalent to the THRESHOLD parameter, which you can set when you run the findpeaks function.

- **Label peaks** — When this check box is selected, the Peak Finder displays the calculated peak values directly on the plot. By default, this check box is cleared and no values are displayed on the plot.

- **Sort by time** — When this check box is selected, the Peak Finder puts the largest calculated peak values in the time range shown in the display in time-sorted order. The Peak Finder displays the peak values in the **Peaks** pane. The **Max Num of Peaks** parameter still controls the number of peaks listed. By default, this check box is cleared and the **Peak Finder** panel displays the largest calculated peak values in the **Peaks** pane in decreasing order of peak height.

Peaks Pane

The **Peaks** pane displays all of the largest calculated peak values and the times at which the peaks occur, using the parameters you define in the **Settings** pane. You set the **Max Num of Peaks** parameter to specify the number of peaks shown in the list. Using the **Sort by time** check box, you set the order in which the peaks are listed.

The numerical values displayed in the **Value** column are equivalent to the pks output argument returned when you run the findpeaks function. The numerical values displayed in the **Time** column are similar to the locs output argument returned when you run the findpeaks function.

When you use the zoom options in the Scope, the peak locations automatically adjust to the time range shown in the display. In the Scope toolbar, click the **Zoom In** or **Zoom X** button to constrict the

x-axis range of the display, and the statistics shown reflect this time range. For example, you can zoom in on one peak to make the **Peak Finder** panel display information about only that particular peak.

Style Dialog Box

In the **Style** dialog box, you can customize the style of displays. You are able to change the color of the figure containing the displays, the background and foreground colors of display axes, and properties of lines in a display. From the Scope menu, select **View > Style** to open this dialog box.

Properties

The **Style** dialog box allows you to modify the following properties of the Scope GUI:

Figure color

Specify the color that you want to apply to the background of the Scope GUI. By default, the figure color is gray.

Plot type

Specify the type of plot to use. The default setting is Line. Valid values for **Plot type** are:

- Line — Displays input signal as lines connecting each of the sampled values. This approach is similar to the functionality of the MATLAB line or plot function.

- Stairs — Displays input signal as a *stairstep* graph. A stairstep graph is made up of only horizontal lines and vertical lines. Each horizontal line represents the signal value for a discrete sample period and is connected to two vertical lines. Each vertical line represents a change in values occurring at a sample. This approach is equivalent to the MATLAB stairs function. Stairstep graphs are useful for drawing time history graphs of digitally sampled data.

- Auto — Displays input signal as a line graph if it is a continuous signal and displays input signal as a stairstep graph if it is a discrete signal.

This parameter is Tunable.

Select display

Specify the active display as a number, where a display number corresponds to the index of the input signal. The number of a display corresponds to its column-wise placement index. The default setting is 1. Set this parameter to control which display should have its axes colors, line properties, marker properties, and visibility changed. Tunable.

Axes colors

Specify the color that you want to apply to the background of the axes for the active display.

Properties for line

Specify the signal for which you want to modify the visibility, line properties, and marker properties.

Visible

Specify whether the selected signal on the active display should be visible. If you clear this check box, the line disappears.

Line

Specify the line style, line width, and line color for the selected signal on the active display.

Marker

Specify marks for the selected signal on the active display to show at data points. This parameter is similar to the Marker property for the MATLAB Handle Graphics plot objects. You can choose any of the marker symbols from the following table.

Specifier	Marker Type
none	No marker (default)
○	Circle
□	Square
×	Cross
●	Point
+	Plus sign
✳	Asterisk
◇	Diamond
▽	Downward-pointing triangle
△	Upward-pointing triangle
◁	Left-pointing triangle
▷	Right-pointing triangle

Specifier	Marker Type
☆	Five-pointed star (pentagram)
✡	Six-pointed star (hexagram)

**Core —
General
UI Options**

The Core:General UI Options dialog box controls the general settings for the Time Scope GUI. From the Time Scope menu bar, select **File > Configuration** to open the Time Scope — Configuration dialog box. Click on the **Core** pane, and the following dialog box appears.

To open the Core:General UI Options dialog box, select **General UI** and click **Options**. The following dialog box appears.

General UI Options

Open message log
Use this parameter to control when the Message Log window opens.
The Message Log window helps you debug any issues with the Time
Scope. Using the drop-down list, you can choose to open the Message
Log window under any of the following conditions:

- `for any new messages`

- `for warn/fail messages`

- `only for fail messages`

- `manually`

You can open the Message Log at any time by selecting **Help > Message
Log**. The Message Log dialog box provides a system-level record of
loaded configuration settings and registered extensions. The Message
Log displays summaries and details of each message, and you can filter
the display of messages by **Type** and **Category**.

- The **Type** parameter allows you to select which types of messages to
  display in the Message Log. You can select `All`, `Info`, `Warn`, or `Fail`.

- The **Category** parameter allows you to select the category of
  messages to display in the Message Log. You can select `All`,
  `Configuration`, or `Extension`.

- The scope uses `Configuration` messages to indicate when new configuration files are loaded.

- The scope uses `Extension` messages to indicate when components are registered. For example, you may see a `Simulink registered` message, indicating that a component has been successfully registered with Simulink and is available for configuration.

Visuals — Time Domain Options

The Time Scope — Visuals:Time Domain Options dialog box controls the visual configuration settings of the Time Scope displays. From the Time Scope menu, select **View > Properties** to open this dialog box.

Main Pane

The **Main** pane of the Time Scope — Visuals:Time Domain Options dialog box appears as follows.

Input processing

Specify whether the Time Scope should treat the input signal as
`Columns as channels (frame based)` or `Elements as channels
(sample based)`. Tunable.

Frame-based processing is only available for discrete input signals. For
more information about frame-based input channels, see the "What Is
Frame-Based Processing?" section in the *DSP System Toolbox User's
Guide*. For an example that uses the Time Scope block and frame-based
input signals, see the Display Time-Domain Data in the Time Scope
section in the *DSP System Toolbox User's Guide*.

Time span

Specify the time span, either by selecting a predefined option or by
entering a numeric value in seconds. You can select one of the following
options:

- `Auto calculate` — In this mode, the Time Scope automatically calculates the appropriate value for time span from the difference between the simulation "Start time" and "Stop time" parameters. This option is available only for the Time Scope block but not for the `dsp.TimeScope` System object.

- `One frame period` — In this mode, the Time Scope uses the frame period of the input signal to the Time Scope block. This option is only available when the **Input processing** parameter is set to `Columns as channels (frame based)`. This option is not available when you set the **Input processing** parameter to `Elements as channels (sample based)`.

- `<user defined>` — In this mode, you specify the time span by replacing the text `<user defined>` with a numeric value in seconds.

The scope sets the *time*-axis limits using the value of this parameter and the value of the **Time display offset** parameter. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the *time*-axis limits as shown in the following figure.

This parameter is Tunable.

Time span overrun

Specify how the scope displays new data beyond the visible time span. The default setting is `Wrap`. You can select one of the following options:

- Wrap — In this mode, the scope displays new data until the data reaches the maximum *time*-axis limit. When the data reaches the maximum *time*-axis limit of the scope window, the Time Scope clears the display. The Time Scope then updates the time offset value, and begins displaying subsequent data points starting from the minimum *time*-axis limit.

- Scroll — In this mode, the scope scrolls old data to the left to make room for new data on the right side of the scope display. This mode is graphically intensive and can affect run-time performance, but it is beneficial for debugging and monitoring time-varying signals.

This parameter is Tunable.

Time units

Specify the units used to describe the *time*-axis. The default setting is Metric. You can select one of the following options:

- Metric — In this mode, Time Scope converts the times on the *time*-axis to some metric units such as milliseconds, microseconds, days, etc. Time Scope chooses the appropriate metric units, based on the minimum *time*-axis limit and the maximum *time*-axis limit of the scope window.

- Seconds — In this mode, Time Scope always displays the units on the *time*-axis as seconds.

- None — In this mode, Time Scope displays no units on the *time*-axis. Time Scope shows only the word Time on the *time*-axis.

This parameter is Tunable.

Time display offset

This parameter allows you to offset the values displayed on the *time*-axis by a specified number of seconds. When you specify a scalar value, the scope offsets all channels equally. When you specify a vector of offset values, the scope offsets each channel independently. Tunable.

When you specify a **Time display offset** vector of length *N*, the scope offsets the input channels as follows:

- When *N* is equal to the number of input channels, the scope offsets each channel according to its corresponding value in the offset vector.

- When *N* is less than the number of input channels, the scope applies the values you specify in the offset vector to the first *N* input channels. The scope does not offset the remaining channels.

- When *N* is greater than the number of input channels, the scope offsets each input channel according to the corresponding value in the offset vector. The scope ignores all values in the offset vector that do not correspond to a channel of the input.

The scope computes the *time*-axis range using the values of the **Time display offset** and **Time span** parameters. For example, if you set the **Time display offset** to `5e-6` and the **Time span** to `25e-6`, the scope sets the *time*-axis limits as shown in the following figure.

Similarly, when you specify a vector of values, the scope sets the minimum *time*-axis limit using the smallest value in the vector. To set the maximum *time*-axis limit, the scope sums the largest value in the vector with the value of the **Time span** parameter. For more information, see "Signal Display" on page 2-1447.

Show time-axis labels

Specify how to display the time units used to describe the *time*-axis. The default setting is `All`. You can select one of the following options:

- All — In this mode, the *time*-axis labels appear in all displays.

- None — In this mode, the *time*-axis labels do not appear in the displays.

- Bottom Displays Only — In this mode, the *time*-axis labels appear in only the bottom row of the displays.

Tunable.

Maximize axes

Specify whether to display the Time Scope in maximized axes mode. In this mode, each of the axes are expanded to fit into the entire display. In each display, there is no space to show labels. Tick mark values are shown on top of the plotted data. The default setting is Auto. You can select one of the following options:

- Auto — In this mode, the axes appear maximized in all displays only if the **Title** and **Y-Axis label** parameters are empty for every display. If you enter any value in any display for either of these parameters, the axes are not maximized.

- On — In this mode, the axes appear maximized in all displays. Any values entered into the **Title** and **Y-Axis label** parameters are hidden.

- Off — In this mode, none of the axes appear maximized.

Display Pane

The **Display** pane of the Visuals:Time Domain Options dialog box appears as follows.

dsp.TimeScope

Select display

Specify the active display as a number, where a display number corresponds to the index of the input signal. The index of a display is its column-wise placement index. The default setting is 1. Set this parameter to control the display for which you intend to modify the title, legend, grid, *y*-axis limits, *y*-axis label, and magnitude/phase display parameters. Tunable.

Title

Specify the active display title as a string. By default, the active display has no title. Tunable.

Show legend

Select this check box to show the legend in the active display. The channel legend displays a name for each channel of each input signal. When the legend appears, you can place it anywhere inside of the scope window. To turn the legend off, clear the **Show legend** check box. Tunable.

You can edit the name of any channel in the legend. To do so, double-click the current name, and enter a new channel name. By default, the scope names each channel according to either its signal name or the name of the block from which it comes. If the signal has multiple channels, the scope uses an index number to identify each channel of that signal. For example, a 2-channel signal named `MySignal` would have the following default names in the channel legend: `MySignal:1`, `MySignal:2`.

The Time Scope does not display signal names that were labeled within an unmasked subsystem. You must label all input signals to the Scope block that originate from an unmasked subsystem.

To change the appearance of any channel of any input signal in the Scope window, from the menu, select **View > Style**.

Show grid
When you select this check box, a grid appears in the active display of the Scope. To hide the grid, clear this check box. Tunable.

Plot signal(s) as magnitude and phase
When you select this check box, the Scope splits the active display into a magnitude plot and a phase plot. By default, this check box is cleared. If the input signal is complex valued, the Scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes within the same active display, as shown in the following figure.

Selecting this check box and clicking the **Apply** or **OK** button changes the active display. The magnitude of the input signal appears on the top axes and its phase, in degrees, appears on the bottom axes. See the following figure.

This feature is particularly useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box always produces the same behavior. The phase is 0 for nonnegative input and 180 degrees for negative input. Tunable.

Minimum Y-limit

Specify the minimum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a minimum value of –180 degrees.

Maximum Y-limit

Specify the maximum value of the *y*-axis. Tunable.

When you select the **Plot signal(s) as magnitude and phase** check box, the value of this parameter always applies to the magnitude plot on the top axes. The phase plot on the bottom axes is always limited to a maximum value of 180 degrees.

Y-axis label

Specify the text for the scope to display to the left of the *y*-axis. Tunable.

This parameter becomes invisible when you select the **Plot signal(s) as magnitude and phase** check box. When you enable that parameter, the *y*-axis label always appears as Magnitude on the top axes and Phase on the bottom axes.

Tools — Plot Navigation Options

The Time Scope — Tools:Plot Navigation Options dialog box provides you with the ability to automatically zoom in on and zoom out of your data, and to scale the axes of the Time Scope. In the Time Scope menu, select **Tools > Axes Scaling Options** to open this dialog box.

Alternatively, you can open this dialog box from the Time Scope — Configuration dialog box. In the Time Scope menu, select **File > Configuration** to open the Time Scope — Configuration dialog box. Click the **Tools** pane, and the following dialog box appears.

The **Tools** pane in the Time Scope – Configuration dialog box contains the tools that appear on the Time Scope GUI. By default, the **Plot Navigation** tool is enabled. If you disable the tool by clearing the **Enabled** check box, the axes scaling and zoom tools no longer display in the Time Scope GUI. You can reenable the tool at any time by selecting the **Enabled** check box.

By default, the **Measurements** tool is also enabled. If you disable the tool by clearing the **Enabled** check box, the Time Domain Measurements tools no longer display in the Time Scope GUI. See the "Measurements Panels" on page 5-1112 section for more information. You can reenable the tool at any time by selecting the **Enabled** check box.

To open the Time Scope — Tools:Plot Navigation Options dialog box, select the **Plot Navigation** tool, and then click the **Options** button.

Parameters

The Tools:Plot Navigation Options dialog box appears as follows.

Axis Scaling

Specify when the Scope should automatically scale the axes. You can select one of the following options:

- Manual — When you select this option, the Scope does not automatically scale the axes. You can manually scale the axes in any of the following ways:

  - Select **Tools > Scale axes limits**.

  - Press the **Scale Axes Limits** toolbar button.

  - When the scope GUI is the active window, press **Ctrl+A**.

- Auto — When you select this option, the Scope scales the axes as needed, both during and after simulation. Selecting this option enables the **Do not allow Y-axis limits to shrink** check box. By default, this parameter is selected, and the Scope does not shrink the *y*-axis limits when scaling the axes.

- Once at stop — Selecting this option causes the Scope to scale the axes each time simulation is stopped. The Scope does not scale the axes during simulation.

This parameter is Tunable.

Do not allow Y-axis limits to shrink

When you select this parameter, the y-axis limits are only allowed to grow during axes scaling operations. If you clear this check box, the y-axis limits may shrink during axes scaling operations.

This parameter appears only when you select Auto for the **Axis Scaling** parameter. When you set the **Axis Scaling** parameter to Manual or Once at stop, the y-axis limits are allowed to shrink. Tunable.

Y-axis Data range (%)

Set the percentage of the y-axis that the Scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to 100, the Scope scales the y-axis limits such that your data uses the entire y-axis range. If you then set this parameter to 30, the Scope increases the y-axis range such that your data uses only 30% of the y-axis range. Tunable.

Y-axis Align

Specify where the Scope should align your data with respect to the y-axis when it scales the axes. You can select Top, Center, or Bottom. Tunable.

Scale X-axis limits

Check this box to allow the Scope to scale the x-axis limits when it scales the axes. Tunable.

X-axis Data range (%)

Set the percentage of the x-axis that the Scope should use to display the data when scaling the axes. Valid values are between 1 and 100. For example, if you set this parameter to 100, the Scope scales the x-axis limits such that your data uses the entire$x$-axis range. If you then set this parameter to 30, the Scope increases the x-axis range such that your data uses only 30% of the x-axis range. Use the x-axis **Align** parameter to specify data placement with respect to the x-axis.

This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

X-axis Align

Specify how the Scope should align your data with respect to the *x*-axis: Left, Center, or Right. This parameter appears only when you select the **Scale X-axis limits** check box. Tunable.

**Sources —
Streaming
Options**

The Sources:Streaming Options dialog box lets you control the number of input signal samples that Time Scope holds in memory. In the Time Scope menu, select **View** > **Data History Options** to open this dialog box.

Data History Options

Data History Options

Limit data points to last: 50000

OK   Cancel   Apply

Limit data points to last

Specify the number of input signal samples that Time Scope holds in memory. The default is 50000.

Examples

The first few examples illustrate how to use the Time Scope block to view a variety of input signals in the time domain.

- "Example: Display Simple Sine Wave Input Signal" on page 5-1154
- "Example: View Sine Wave Input Signals at Different Sample Rates and Offsets" on page 5-1154
- "Example: Display Complex-valued Input Signal" on page 5-1155
- "Example: Display Variable-size Input Signal" on page 5-1157

The remaining examples demonstrate how to use the Measurements Panels in the Time Scope GUI to glean information about the input signals.

- "Example: Use Bilevel Measurements Panel with Clock Input Signal" on page 5-1158
- "Example: Find Heart Rate Using Peak Finder Panel with ECG Input Signal" on page 5-1161

Example: Display Simple Sine Wave Input Signal

Construct dsp.SineWave and dsp.TimeScope objects. Run the step method to display the signal.

```
hsin = dsp.SineWave('Frequency',100, 'SampleRate', 1000);
hsin.SamplesPerFrame = 10;
hts1 = dsp.TimeScope('SampleRate', hsin.SampleRate,'TimeSpan', 0.1);
for ii = 1:10
    x = step(hsin);
    step(hts1, x);
end
```

Run the release method to automatically scale the axes.

```
release(hts1)
```

Run the MATLAB clear function to close the Time Scope window.

```
clear hts1 hsin x;
```

Example: View Sine Wave Input Signals at Different Sample Rates and Offsets

Construct dsp.SineWave and dsp.FirDecimator objects. Use the dsp.FirDecimator object to create a new signal that equals the original signal, but decimated by a factor of 2. Construct a dsp.TimeScope object with 2 input ports. Run the step method to display the signal.

```
Fs = 1000;  % Sampling frequency
hsin1 = dsp.SineWave('Frequency',50,...
   'SampleRate',Fs, ...
   'SamplesPerFrame', 100);
% Create FIRDecimator System object to decimate by 2
```

```
hfilt = dsp.FIRDecimator;
% Create TimeScope System object with 2 input ports (channels)
hts2 = dsp.TimeScope(2, [Fs Fs/2], ...
    'TimeDisplayOffset', [O 38/Fs], ...
    'TimeSpan', 0.25, ...
    'YLimits',[-1 1], ...
    'LegendSource', 'Auto');
for ii = 1:2
     xsine = step(hsin1);
     xdec = step(hfilt,xsine);
     step(hts2, xsine, xdec);
end
```

Run the `release` method to automatically scale the axes.

```
release(hts2)
```

Run the MATLAB `clear` function to close the Time Scope window.

```
clear hts2 Fs hsin1 hfilt ii xsine xdec;
```

Example: Display Complex-valued Input Signal

Create a vector representing a complex-valued sinusoidal signal and construct a `dsp.TimeScope` object. Run the `step` method to display the signal.

```
fs = 1000; t = (0:1/fs:10)';
CxSine = cos(2*pi*0.2*t) + 1i*sin(2*pi*0.2*t);
CxSineSum = cumsum(CxSine);
figure(101); subplot(2,1,1); stairs(t,abs(CxSineSum));  % Plot magnitu
subplot(2,1,2); stairs(t,(180/pi)*angle(CxSineSum));    % Plot phase :
h1 = dsp.TimeScope(1, fs, 'TimeSpanSource', 'Auto');
step(h1,CxSineSum);
```

By default, when the input is a complex-valued signal, Time Scope plots the real and imaginary portions on the same axes. These real and imaginary portions appear as different-colored lines on the same axes within the same active display, as shown in the following figure.

Change the `MagnitudePhase` property to true.

```
set(h1,'MagnitudePhase',true);
```

Time Scope now plots the magnitude and phase of the input signal on two separate axes within the same active display. The active display changes to show the magnitude of the input signal on the top axes and its phase, in degrees, on the bottom axes, as shown in the following figure.

Run the release method to automatically scale the axes.

```
release(h1);
```

Run the MATLAB clear function to close the Time Scope window, and remove the variables you created from the workspace.

```
close(101);
clear h1 fs t CxSine CxSineSum
```

Example: Display Variable-size Input Signal

Create a vector that represents a two-channel constant signal. Create another vector that represents a three-channel constant signal. Construct a dsp.TimeScope object with 2 input ports. Run the step method to display the signal.

```
fs = 10;
```

```
sigdim2 = [ones(5*fs,1), 1+ones(5*fs,1)];                          % 2-dim 0-5s
sigdim3 = [2+ones(5*fs,1), 3+ones(5*fs,1), 4+ones(5*fs,1)]; % 3-dim 5-10s
h2 = dsp.TimeScope(2, fs, 'TimeSpanSource', 'Property');
set(h2,'PlotType','Stairs');
set(h2,'TimeSpanOverrunMode','Scroll');
set(h2,'TimeDisplayOffset',[0,0,5]);
step(h2,[sigdim2; sigdim3(:,1:2)], sigdim3(:,3));
```

In this example, the size of the input signal to the Time Scope block changes as the simulation progresses. When the simulation time is less than 5 seconds, Time Scope plots only the signal *sigdim2*, which has two channels. After 5 seconds, Time Scope also plots the signal *sigdim3*, which has three channels.

Run the `release` method to automatically scale the axes.

```
release(h2);
```

Run the MATLAB `clear` function to close the Time Scope window, and remove the variables you created from the workspace.

```
clear h2 fs sigdim2 sigdim3
```

Example: Use Bilevel Measurements Panel with Clock Input Signal

Load the clock data into the variable, *x*. Find the sample time, *ts*.

```
load clockex;
ts = t(2)-t(1);
```

Construct a `dsp.TimeScope` object. Run the `step` method to display the signal.

```
h4 = dsp.TimeScope(1, 1/ts, 'TimeSpanSource','Auto');
step(h4,x);
```

Run the `release` method to automatically scale the axes.

```
release(h4);
```

To show the **Bilevel Measurements** panel, in the Time Scope menu, select **Tools > Measurements > Bilevel Measurements**. To collapse the **Transitions** pane, click the pane collapse button (▼) next to that label. To expand the **Settings** pane and the **Overshoots / Undershoots** pane, click the pane expand button (▶) next to each label. The Time Scope figure appears as shown in the following figure.

As you can see in the figure, the value for the rising edge **Settling Time** parameter is initially not displayed. The reason for this is that the default value for the **Settle Seek** parameter is too large for this example. In this case, the settle seek time is longer than the entire simulation duration. Enter a value for settle seek of `2e-6`, and press the **Enter** key. Time Scope now displays a rising edge settling time value of `118.392 ns`.

This settling time value displayed is actually the statistical average of the settling times for all five rising edges. To show the settling time for only one rising edge, you can zoom in on that transition. In the Time

Scope toolbar, click the Zoom X button (ⁱᵂQⁱ). Then, click the display near a value of 2 microseconds on the *time*-axis. Drag to the right and release near a value of 4 microseconds on the *time*-axis. Time Scope updates the rising edge **Settling Time** value to reflect the new time window, as shown in the following figure.

Run the MATLAB `clear` function to close the Time Scope window, and remove the variables you created from the workspace.

```
clear h4 x t ts
```

Example: Find Heart Rate Using Peak Finder Panel with ECG Input Signal

First, create the electrocardiogram (ECG) signal.

```
x1 = 3.5*ecg(2700).';
y1 = sgolayfilt(kron(ones(1,13),x1),0,21);
n = (1:30000)';
del = round(2700*rand(1));
mhb = y1(n + del);
ts = 0.00025;
```

This example uses the Savitzky-Golay filter in Signal Processing Toolbox. For more information, see the sgolayfilt function reference page or run the sgolaydemo example.

Construct a dsp.TimeScope object. Run the step method to display the signal.

```
h5 = dsp.TimeScope(1, 1/ts, 'TimeSpanSource','Auto');
step(h5,mhb);
```

Run the release method to automatically scale the axes.

```
release(h5);
```

To show the **Peak Finder** panel, in the Time Scope menu, select **Tools > Measurements > Peak Finder**. To expand the **Settings** pane, click the pane expand button (▶) next to that label. Enter a value for **Max Num of Peaks** of 10, and press the **Enter** key. Time Scope now displays in the **Peaks** pane a list of 10 peak amplitude values, and the times at which they occur, as shown in the following figure.

As you can see from the list of peak values, there is a constant time difference of 0.675 seconds between each heartbeat. Therefore, the heart rate of the ECG signal is given by the following equation.

$$\frac{60\frac{\sec}{\min}}{0.675\frac{\sec}{\text{beat}}} = 88.89\frac{\text{beats}}{\min}(\text{bpm})$$

Run the MATLAB clear function to close the Time Scope window, and remove the variables you created from the workspace.

```
clear h5 x1 y1 n del mhb ts
```

See Also   Time Scope | dsp.SpectrumAnalyzer | sptool

Purpose   Create time scope object with same property values

Syntax   `C = clone(H)`

Description   `C = clone(H)` creates a `TimeScope` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

dsp.TimeScope.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns the number of expected inputs, `N`, to the `step` method. `N` equals the value of the `NumInputPorts` property.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method. The time scope is a sink object, so N equals 0.

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

dsp.TimeScope.hide

Purpose   Hide time scope window

Syntax   hide(H)

Description   hide(H) hides the time scope window, associated with System object, H.

See Also   dsp.TimeScope.show

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the TimeScope object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.TimeScope.release

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description release(H) releases system resources, such as memory, file handles,
and hardware connections. This method lets you change any properties
or input characteristics.

You should call the release method after calling the step method when
there is no new data for the simulation. When you call the release
method, the axes will automatically scale in the Time Scope figure
window. After calling the release method, any non-tunable properties
can be set once again.

Note You can use the release method on a System object in code
generated from MATLAB, but once you release its resources, you cannot
use that System object again.

Examples   View a sine wave on the time scope. When you finish the simulation,
release system resources. Note that when you run the release method,
the axes automatically scale.

```
hsin = dsp.SineWave('Frequency',100, 'SampleRate', 1000);
hsin.SamplesPerFrame = 10;

hts1 = dsp.TimeScope('SampleRate', hsin.SampleRate,'TimeSpan', 0.1);
for ii = 1:10
    x = step(hsin);
    step(hts1, x);
end

release(hts1);
```

View two sine waves with different sample rates and time offsets. When you finish the simulation, release system resources. Note that when you run the release method, the axes automatically scale.

```
Fs = 1000;  % Sampling frequency

hsin1 = dsp.SineWave('Frequency',50,...
   'SampleRate',Fs, ...
   'SamplesPerFrame', 100);

% Create FIRDecimator System object to decimate by 2
hfilt = dsp.FIRDecimator;

% Create TimeScope System object with 2 input
% ports (channels)
hts2 = dsp.TimeScope(2, [Fs Fs/2], ...
   'TimeDisplayOffset', [0 38/Fs], ...
   'TimeSpan', 0.25, ...
   'YLimits',[-1 1], ...
   'LegendSource', 'Auto');

for ii = 1:2
    xsine = step(hsin1);
    xdec = step(hfilt,xsine);
    step(hts2, xsine, xdec);
end

release(hts2);
```

Algorithms   In operation, the release method is similar to the mdlTerminate function.

See Also   dsp.TimeScope.reset

dsp.TimeScope.reset

Purpose   Reset internal states of time scope object

Syntax   reset(H)

Description   reset(H) sets the internal states of the TimeScope object H to their
                initial values.

                You should call the reset method after calling the step method when
                you want to clear the Time Scope figure displays, prior to releasing
                system resources. This action enables you to start a simulation from the
                beginning. When you call the reset method, the displays will become
                blank again. In this sense, its functionality is similar to that of the
                MATLAB clf function. Do not call the reset method after calling the
                release method.

Examples   View a sine wave on the time scope. When you finish the simulation,
                pause for 5 seconds and then reset the screen. Then, view a different
                sine wave on the same time scope.

```
hsin = dsp.SineWave('Frequency', 100, 'SampleRate', 1000, 'SamplesPerFram
hsin1 = dsp.SineWave('Frequency', 50, 'SampleRate', 1000, 'SamplesPerFram

hts1 = dsp.TimeScope('SampleRate', hsin.SampleRate,'TimeSpan', 0.1);
for ii = 1:10
    x = step(hsin);
    step(hts1, x);
end

% Pause and then reset internal states
pause(5);
reset(hts1);

for ii = 1:10
    x = step(hsin1);
    step(hts1, x);
end
```

```
release(hts1);
```

Algorithms   In operation, the reset method is similar to a consecutive execution of the mdlTerminate function and the mdlInitializeConditions function.

See Also   dsp.TimeScope | dsp.TimeScope.release

dsp.TimeScope.show

Purpose   Make time scope window visible

Syntax   show(H)

Description show(H) makes the time scope window, associated with System object, H, visible.

See Also   dsp.TimeScope.hide

Purpose   Display signal in time scope figure

Syntax   `step(H,X)`
`step(H,X1,X2,...,XN)`

Description   `step(H,X)` displays the signal, `X`, in the time scope figure.

`step(H,X1,X2,...,XN)` displays the signals `X1`, `X2`,...,`XN` in the time scope figure when you set the `NumInputPorts` property to `N`. In this case, `X1`, `X2`,...,`XN` can have different data types and dimensions.

dsp.TransitionMetrics

Purpose
Transition metrics of bilevel waveforms

Description
The `TransitionMetrics` object extracts information such as duration, slew rate, and reference-level crossings for each transition found in the bilevel waveform. The `TransitionMetrics` object can additionally return preshoot, postshoot and settling metrics for the regions immediately before and after each transition.

To obtain transition metrics for a bilevel waveform:

1 Define and set up your transition metrics. See "Construction" on page 5-1174.

2 Call `step` to calculate the transition metrics according to the properties of `dsp.TransitionMetrics`. The behavior of `step` is specific to each object in the toolbox.

Construction
`H = dsp.TransitionMetrics` creates a transition metrics System object, `H`. The object computes the rise time, fall time and width of a pulse. `TransitionMetrics` additionally computes cycle metrics such as pulse separations, periods, and duty cycles.

`H = dsp.TransitionMetrics('`*PropertyName*`',`*PropertyValue*`,...)` returns a `TransitionMetrics` System object, `H`, with each specified property set to the specified value.

Properties

MaximumRecordLength

Maximum samples to preserve between calls to `step`. This property requires a positive integer that specifies the maximum number of samples to save between calls to the `step` method. When the number of samples to be saved exceeds this length, the oldest excess samples are discarded. This property applies when `RunningMetrics` is `true` and is tunable.

Default: 1000

PercentReferenceLevels

Lower-, middle-, and upper-percent reference levels. This property contains a 3-element numeric row vector that contains the lower-, middle-, and upper-percent reference levels. These reference levels are used as an offset between the low and high states of the waveform when computing the duration of each transition.

Default: [10 50 90]

PercentStateLevelTolerance

Tolerance of the state level (in percent). This property requires a scalar that specifies the maximum deviation from either the low or high state before it is considered to be outside that state. The tolerance is expressed as a percentage of the waveform amplitude.

Default: 2

PostshootOutputPort

Enable posttransition aberration metrics. If this property is set to true, overshoot and undershoot metrics are reported for a region defined immediately after each transition. The posttransition aberration region is defined as the waveform interval that begins at the end of each transition and whose duration is the value of PostshootSeekFactor times the computed transition duration. If a complete subsequent transition is detected before the interval is over, the region is truncated at the start of the subsequent transition. The metrics are computed for each transition that has a complete posttransition aberration region.

Default: false

PostshootSeekFactor

Corresponds to the duration of time to search for the overshoot and undershoot metrics immediately following each transition. The duration is expressed as a factor of the duration of the transition.

This property is enabled only when the `PostshootOutputPort` property is set to `true` and is tunable.

Default: 3

PreshootOutputPort

Enable pretransition aberration metrics. If the `PreshootOutputPort` property is set to `true`, overshoot and undershoot metrics are reported for a region defined immediately before each transition. The pretransition aberration region is defined as the waveform interval that ends at the start of each transition and whose duration is `PreshootSeekFactor` times the computed transition duration.

Default: `false`

PreshootSeekFactor

Corresponds to the duration of time to search for the overshoot and undershoot metrics immediately preceding each transition. The duration is expressed as a factor of the duration of the transition. This property is enabled only when the `PreshootOutputPort` property is set to `true` and is tunable.

Default: 3

RunningMetrics

Enable metrics over all calls to `step`. If `RunningMetrics` is set to `false`, metrics are computed for each call to `step` independently. If `RunningMetrics` is set to `true`, metrics are computed across subsequent calls to `step`. If there are not enough samples to compute metrics associated with the last transition, posttransition aberration region, or settling seek duration in the current record, the object defers reporting all transition, aberration, and settling metrics associated with the last transition until a subsequent call

to `step` is made with enough data to compute all enabled metrics for that transition.

Default: `false`

SampleRate

Sampling rate of uniformly-sampled signal. Specify the sample rate in hertz as a positive scalar. This property is used to construct the internal time values that correspond to the input sample values. Time values start with zero. This property applies when the `TimeInputPort` property is set to `false`.

Default: `1`

SettlingOutputPort

Enable settling metrics. If `SettlingOutputPort` is set to true, settling metrics are reported for each transition. The region used to compute the settling metrics starts at the midcrossing and lasts until the `SettlingSeekDuration` has elapsed. If an intervening transition occurs, or the signal has not settled within the `PercentStateLevelTolerance` of the final level, `NaN` is returned for each metric. If there are not enough samples after the last transition to complete the `SettlingSeekDuration`, no metrics are reported for the last transition. The metrics are reported for the transition the next time `step` is called if the `RunningMetrics` property is set to `true`.

Default: `false`

SettlingSeekDuration

Duration of time over which to search for settling. This property is a scalar that specifies the amount of time to inspect from the mid-reference level crossing (in seconds). If the transition has not yet settled, or a subsequent complete transition is detected within this duration, the `TransitionMetrics` object reports `NaN` for all

settling metrics. This property is tunable and applies only when you set the `SettlingOutputPort` property to `true`.

Default: `0.02`

StateLevels

Low- and high-state levels. This property is a 2-element numeric row vector that contains the low and high state levels respectively. These state levels correspond to the nominal logic low and high levels of the pulse waveform. This property is tunable.

Default: `[0 2.3]`

StateLevelsSource

Auto or manual state level computation. If the `StateLevelsSource` property is set to `'Auto'`, the first record sent to `step` is sent to `dsp.StateLevels` with the default settings to determine the state levels of the incoming waveform. If this property is set to `'Property'`, the object uses the values the user specifies in the `StateLevels` property.

Default: `'Property'`

TimeInputPort

Add input to specify sample instants. Set `TimeInputPort` to `true` to enable an additional real input column vector to `step` to specify the sample instants that correspond to the sample values. If this property is `false`, the sample instants are built internally. The sample instants start at zero and increment by the reciprocal of the `SampleRate` property for subsequent samples. The sample instants continue to increment if the `RunningMetrics` property is set to `true` and no intervening calls to the `reset` or `release` methods are encountered.

Default: `false`

Methods

clone	Clones the current instance of the transition metrics object
getNumInputs	Number of expected inputs to the method
getNumOutputs	Number of outputs of the method
isLocked	Locked status (logical) for input attributes and nontunable properties
plot	Plots signal and metrics computed in last call to
reset	Reset the running transition metrics object
step	Transition metrics of bilevel waveform

Examples

Transition and Preshoot Information of a 2.3 V Step Waveform

Compute transition and preshoot information of a 2.3 V step waveform sampled at 4 MHz

Load the data.

```
load('transitionex.mat', 'x');
```

Construct your transition metrics System object. Set the SampleRate property to 4 MHz, set the StateLevelsSource property to 'Auto' to estimate the state levels from the data, set the PreshootOutputPort property to true to output pretransition aberration metrics when you call step.

```
htm1 = dsp.TransitionMetrics('SampleRate',4e6, ...
                             'StateLevelsSource','Auto', ...
                             'PreshootOutputPort',true)
```

Call `step` to compute the transition and preshoot information. Plot the result.

```
[transition, preshoot] = step(htm1, x)
plot(htm1)
```

Transition, Postshoot, and Settling Information of a 2.3 V Step Waveform

Compute transition, postshoot, and settling information of a 2.3 V step waveform sampled at 4 MHz.

Load the data along with the sampling instants.

```
load('transitionex.mat', 'x', 't');
```

Construct your transition metrics object setting the `TimeInputPort` property to `true` and the `StateLevels` property to match waveform state levels. To output the postshoot information and settling information when you call `step`, set the `PostShootOutputPort` and `SettlingOutputPort` properties to `true`. Set the settling seek duration to 2 microseconds.

```
htm2 = dsp.TransitionMetrics('TimeInputPort',true, ...
                             'StateLevels',[O 2.3], ...
                             'PostshootOutputPort',true, ...
                             'SettlingOutputPort',true, ...
                             'SettlingSeekDuration',2e-6)
```

Compute the transition, postshoot, and settling information with `step` and plot the result.

```
[transition, postshoot, settling] = step(htm2,x,t)
plot(htm2)
```

References   [1] *IEEE Standard on Transitions, Pulses, and Related Waveforms*, IEEE Standard 181, 2003.

See Also   dsp.StateLevels **|** dsp.PulseMetrics **|**

Purpose   Clones the current instance of the transition metrics object

Syntax   clone(H)

Description   clone(H) clones the current instance of the transition metrics object H.

dsp.TransitionMetrics.getNumInputs

Purpose   Number of expected inputs to the `step` method

Syntax   `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns the number of expected inputs, `N`, to the `step` method for the transition metrics object `H`.

The `getNumInputs` method returns a positive integer that is the number of expected inputs (not counting the object itself) to the `step` method. This value will change if you alter any properties that turn inputs on or off. You must call the `step` method with the number of input arguments equal to the result of `getNumInputs(H)`.

Purpose   Number of outputs of the `step` method

Syntax   `N = getNumOutputs(H)`

Description   `N = getNumOutputs(H)` returns the number of outputs, `N`, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status (logical) for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the transition metrics object H.

Purpose   Plots signal and metrics computed in last call to `step`

Syntax   `plot(H)`

Description   `plot(H)` plots the signal and metrics resulting from the last call of the step method.

By default `plot` displays

- the low and high state levels and the state-level boundaries defined by the `PercentStateLevelTolerance` property.

- the lower-, middle-, and upper-reference levels.

- the locations of the mid-reference level crossings of the positive (+) and negative (-) transitions of each detected pulse.

When you set the `TransitionOutputPort` property to `true`, the locations of the upper and lower crossings are also plotted. When the `PreshootOutputPort` or `PostShootOutputPort` properties are set to `true`, the corresponding overshoots and undershoots are plotted as inverted or non-inverted triangles. When the `SettlingOutputPort` property is set to `true`, the locations where the signal enters and remains within the upper- and lower-state boundaries over the specified seek duration are plotted.

Purpose   Reset the running transition metrics object

Syntax   reset(H)

Description

reset(H) clears information from previous calls to step.

Purpose   Transition metrics of bilevel waveform

Syntax
```
PULSE = step(H,X)
[PULSE,CYCLE] = step(H,X)
[PULSE,TRANSITION] = step(H,X)
[PULSE,PRESHOOT] = step(H,X)
[PULSE,POSTSHOOT] = step(H,X)
[PULSE,SETTLING] = step(H,X)
[...] = step(H,X,T)
```

Description   PULSE = step(H,X) returns a structure array, PULSE, whose fields
contain real-valued column vectors. The number of rows of each field
corresponds to the number of complete pulses found in the real-valued
column vector input, X. Each pulse starts with a transition of the
polarity specified by the Polarity property and ends with a transition
of the opposite polarity.

PULSE fields:

- PositiveCross — Instants where the positive-going transitions cross
  the mid-reference level of each pulse

- NegativeCross — Instants where the negative-going transitions
  cross the mid-reference level of each pulse

- Width — Absolute difference between PositiveCross and
  NegativeCross of each pulse

- RiseTime — Duration between the linearly-interpolated instants
  when the positive-going (rising) transition of each pulse crosses the
  lower- and upper-reference levels

- FallTime — Duration between the linearly-interpolated instants
  when the negative-going (falling) transition of each pulse crosses the
  upper- and lower-reference levels

[PULSE,CYCLE] = step(H,X) returns a structure array, CYCLE,
whose fields contain real-valued column vectors when you set the
CycleOutputPort property to true. The number of rows of each field

corresponds to the number of complete pulse periods found in the real-valued column vector input, X. You need at least three consecutive alternating polarity transitions that start and end with the same polarity as the value of the `Polarity` property if you want to compute cycle metrics. If the last transition found the input, X, does not match the polarity of the `Polarity` property, the pulse separation, period, frequency, and duty cycle are not reported for the last pulse. If the `RunningMetrics` property is set to `true` when this occurs, all pulse, cycle, transition, preshoot, postshoot, and settling metrics associated with the last pulse are deferred until a subsequent call to `step` detects the next transition.

`CYCLE` fields:

- `Period` — Duration between the first transition of the current pulse and the first transition of the next pulse

- `Frequency` — Reciprocal of the period

- `Separation` — Durations between the mid-reference level crossings of the second transition of each pulse and the first transition of the next pulse

- `Width` — Durations between the mid-reference level crossings of the first and second transitions of each pulse. This is equivalent to the width parameter of the `PULSE` structure.

- `DutyCycle` — Ratio of the width to the period for each pulse

`[PULSE,TRANSITION] = step(H,X)` returns a structure array, `TRANSITION`, when you set the `TransitionOutputPort` property to `true`. The fields of `TRANSITION` contain real-valued matrices with two columns which correspond to the metrics of the first and second transitions. The number of rows corresponds to the number of pulses found in the input waveform.

`TRANSITION` fields:

- `Duration` — Amount of time between the interpolated instants where the transition crosses the lower- and upper-reference levels

- `SlewRate` — Ratio of absolute difference between the upper- and lower-reference levels to the transition duration

- `MiddleCross` — Linearly-interpolated instant where the transition first crosses the mid-reference level

- `LowerCross` — Linearly-interpolated instant where the signal crosses the lower-reference level

- `UpperCross` — Linearly-interpolated instant where the signal crosses the upper-reference level

`[PULSE,PRESHOOT] = step(H,X)` returns a structure array, `PRESHOOT`, when you set the `PreshootOutputPort` property to `true`. The fields of `PRESHOOT` contain real-valued 2-column matrices whose row length corresponds to the number of transitions found in the input waveform. The field names are identical to those of the `POSTSHOOT` structure.

`[PULSE,POSTSHOOT] = step(H,X)` returns a structure array, `POSTSHOOT`, when you set the `PostshootOutputPort` property to `true`. The fields of `POSTSHOOT` contain real-valued 2-column matrices whose row length corresponds to the number of transitions found in the input waveform.

`PRESHOOT` and `POSTSHOOT` fields:

- `Overshoot` — Overshoot of the region of interest expressed as a percentage of the waveform amplitude

- `Undershoot` — Undershoot of the region of interest expressed as a percentage of the waveform amplitude

- `OvershootLevel` — Level of the overshoot

- `UndershootLevel` — Level of the undershoot

- `OvershootInstant` — Instant that corresponds to the overshoot

- `UndershootInstant` — Instant that corresponds to the undershoot

`[PULSE,SETTLING] = step(H,X)` returns a structure, `SETTLING`, when you set the `SettlingOutputPort` property to `true`. The fields of

SETTLING correspond to the settling metrics for each transition. Each field is a column vector whose elements correspond to the individual settling durations, levels, and instants.

SETTLING fields:

- Duration — Amount of time from when the signal crosses the mid-reference level to the time where the signal enters and remains within the specified PercentStateLevelTolerance of the waveform amplitude over the specified settling seek duration

- Instant — Instant in time where the signal enters and remains within the specified tolerance

- Level — Level of the waveform where it enters and remains within the specified tolerance

The above operations can be used simultaneously, provided the System object properties are set appropriately. One example of providing all possible inputs and returning all possible outputs is :

[PULSE,CYCLE,TRANSITION,PRESHOOT,POSTSHOOT,SETTLING] = step(H,X) which returns the PULSE, CYCLE, TRANSITION, PRESHOOT, POSTSHOOT, and SETTLING structure arrays when the CycleOutputPort, PreshootOutputPort, PostshootPort, and SettlingOutputPort properties are true. You may enable or disable any combination of output ports. However, the output arguments are defined in the order shown here.

[...] = step(H,X,T) performs the above metrics with respect to a sampled signal, whose sample values, X, and sample instants, T, are real-valued column vectors of the same length. The additional input T applies only when you set the TimeInputPort property to true.

Purpose   Receive UDP packets from network

Description   The UDPReceiver object receives UDP packets from the network.

To receive UDP packets from the network:

1 Define and set up your UDP receiver. See "Construction" on page 5-1191.

2 Call step to receive the UDP packets according to the properties of dsp.UDPReceiver. The behavior of step is specific to each object in the toolbox.

Construction H = dsp.UDPReceiver returns a UDP receiver System object, H, that receives UDP packets from a specified port.

H = dsp.UDPReceiver('*PropertyName*',*PropertyValue*, ...) returns a window object, H, with each property set to the specified value.

Properties   **LocalIPPort**

Local port on which to receive data

Specify the port on which to receive data. The default is 25000.

RemoteIPAddress

Address from which data was sent

Specify the remote IP address from which to receive data. The default is '0.0.0.0', which indicates that data is accepted from any remote IP address.

ReceiveBufferSize

Maximum size of internal buffer

Specify the size of the buffer that receives UDP packets, in bytes. The default is 8192.

MaximumMessageLength

Maximum size of output message

Specify the size of the output message. If received packets exceed the specified value, their contents are truncated. The default is 255.

MessageDataType

Data type of the message

Specify the data type of the message as | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | logical |. The default is uint8.

Methods

clone	Create UDP receiver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Multiply input by window

Examples

Send some UDP packets, and calculate the number of successfully transmitted bytes:

```
hudpr = dsp.UDPReceiver;
hudps = dsp.UDPSender;

bytesSent = 0;
bytesReceived = 0;
dataLength = 128;
```

```
for k = 1:20
   dataSent = uint8(255*rand(1,dataLength));
   bytesSent = bytesSent + dataLength;
   step(hudps, dataSent);
   dataReceived = step(hudpr);
   bytesReceived = bytesReceived + length(dataReceived);
end

release(hudps);
release(hudpr);

fprintf('Bytes sent:     %d\n', bytesSent);
fprintf('Bytes received: %d\n', bytesReceived);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on
the UDP Receive block reference page. The object properties correspond
to the block parameters, except:

- The **Output variable-size signal** parameter is not included in the
  System object.

- The **Sample time** parameter is not included in the System object.

See Also   dsp.UDPSender

Purpose   Create UDP receiver object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a UDP receiver object, C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                  the step method.

                  The getNumInputs method returns a positive integer that is the number
                  of expected inputs (not counting the object itself) to the step method.
                  This value will change if you alter any properties that turn inputs on or
                  off. You must call the step method with the number of input arguments
                  equal to the result of getNumInputs(*H*).

dsp.UDPReceiver.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step
                method.

                The getNumOutputs method returns a positive integer that is the
                number of outputs from the step method. This value will change if you
                alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   Y = isLocked(H)

Description   Y = isLocked(H) returns the locked state, Y, of the UDP receiver object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.UDPReceiver.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections. This method lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose Multiply input by window

Syntax P = step (H)

Description

P = step (H) receives one UDP packet, P, from the network.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

The first time you call the step method on a UDPReceiver object, the object also allocates resources and begins listening for data. As a result, the first step call may not receive data.

dsp.UDPSender

Purpose   Send UDP packets to network

Description The UDPSender object sends UDP packets to the network.

To send UDP packets to the network:

1 Define and set up your UDP sender. See "Construction" on page 5-1200.

2 Call step to send the packets according to the properties of dsp.UDPSender. The behavior of step is specific to each object in the toolbox.

Construction H = dsp.UDPSender returns a UDP sender object, H, that sends UDP packets to a specified port.

H = dsp.UDPSender('*PropertyName*',*PropertyValue*, ...) returns a UDP sender object, H, with each property set to the specified value.

Properties   **RemoteIPAddress**

Remote address to which to send data

Specify the remote (i.e., host) IP address to which the data is sent. The default is '127.0.0.1', which is the local host.

RemoteIPPort

Remote port to which to send data

Specify the port at the remote IP address to which the data is sent. The default is 25000.

LocalIPPortSource

Source of the LocalIPPort property

Specify how to determine the port on the host as | Auto | Property |. If you specify Auto, the object selects the port dynamically from the available ports. If you specify Property, the object uses the source specified in the LocalIPPort property. The default is Auto.

LocalIPPort

Local port from which to send data

Specify the port from which to send data. This property applies when you set the LocalIPPortSource property to Auto. The default is 25000.

Methods

clone	Create UDP sender object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Send a UDP packet to network

Examples

Send a number of UDP packets, and calculate the number of successfully transmitted bytes:

```
hudpr = dsp.UDPReceiver;
hudps = dsp.UDPSender;

bytesSent = 0;
bytesReceived = 0;
dataLength = 128;

for k = 1:20
   dataSent = uint8(255*rand(1,dataLength));
   bytesSent = bytesSent + dataLength;
   step(hudps, dataSent);
   dataReceived = step(hudpr);
```

```
      bytesReceived = bytesReceived + length(dataReceived);
end

release(hudps);
release(hudpr);

fprintf('Bytes sent:     %d\n', bytesSent);
fprintf('Bytes received: %d\n', bytesReceived);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the UDP Send block reference page. The object properties correspond to the block parameters.

See Also   dsp.UDPReceiver

Purpose   Create UDP sender object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a UDP sender object, C, with the same property
                values as H. The clone method creates a new unlocked object.

dsp.UDPSender.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   getNumInputs(H)

Description getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the UDP sender object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   Y = release(H)

Description   Y = release(H) releases system resources, such as memory, file
                 handles, and hardware connections. This method lets you change any
                 properties or input characteristics.

Note You can use the release method on a System object in code
generated from MATLAB, but once you release its resources, you cannot
use that System object again.

Purpose   Send a UDP packet to network

Syntax   Y = step(H,PACKET)

Description   Y = step(H,PACKET) sends one UDP packet, PACKET, to the network.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

Purpose   Decode integer input into floating-point output

Description   The UniformDecoder object decodes integer input into floating-point output. The decoder adheres to the definition for uniform decoding specified in ITU-T Recommendation G.701.

To decode an integer input into a floating-point output:

1 Define and set up your uniform decoder. See "Construction" on page 5-1209.

2 Call step to decode the input according to the properties of dsp.UniformDecoder. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.UniformDecoder returns a uniform decoder, H, that performs the inverse operation of the UniformEncoder object, reconstructing quantized floating-point values from encoded integer input.

H = dsp.UniformDecoder('*PropertyName*',*PropertyValue*,...) returns a uniform decoder, H, with each specified property set to the specified value.

H = dsp.UniformDecoder(peakvalue,numbits, '*PropertyName*',*PropertyValue*,...) returns a uniform decoder, H, with the PeakValue property set to peakvalue, the NumBits property set to numbits, and other specified properties set to the specified values.

Properties   **PeakValue**

Largest amplitude represented in encoded input

Specify the largest amplitude represented in the encoded input as a nonnegative numeric scalar. To correctly decode values encoded with the UniformEncoder object, set the PeakValue property in both objects to the same value. For more information on setting this property, see the PeakValue property description on the dsp.UniformEncoder reference page. The default is 1.

NumBits

Number of input bits used to encode data

Specify the number of bits used to encode the input data as an integer value between 2 and 32. The value of this property can be less than the total number of bits supplied by the input data type. To correctly decode values encoded with the UniformEncoder object, set the NumBits property in both objects to the same value. For more information on setting this property, see the NumBits property description on the dsp.UniformEncoder reference page. The default is 3.

OverflowAction

Action to take when integer input is out of range

Specify the behavior of the uniform decoder when the integer input is out of range as Saturate or Wrap. The value of the NumBits property specifies the representable range of the input. The default is Saturate.

OutputDataType

Output data type as single or double

Specify the data type of the output as single or double. The default is double.

Methods

clone	Create uniform decoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Decode integer input into quantized floating-point output

Examples   Decode an encoded sequence:

```
hue = dsp.UniformEncoder;
hue.PeakValue = 2;
hue.NumBits = 4;
hue.OutputDataType = 'Signed integer';
x = (0:0.25:2)'; % Create an input sequence
hud = dsp.UniformDecoder;
hud.PeakValue = 2;
hud.NumBits = 4;
x_encoded = step(hue, x);
% Check that the last element has been saturated.
x_decoded = step(hud, x_encoded);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Uniform Decoder block reference page. The object properties correspond to the block parameters.

See Also   dsp.UniformEncoder

dsp.UniformDecoder.clone

Purpose Create uniform decoder object with same property values

Syntax C = clone(H)

Description C = clone(H) creates a UniformDecoder object C, with the same property values as H. The clone method creates a new unlocked object.

The clone method creates an instance of an object. The property values, but not internal states, are copied into the new instance of the object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.UniformDecoder.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the UniformDecoder object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Decode integer input into quantized floating-point output

Syntax   Y = step(H,X)

Description   Y = step(H,X) reconstructs quantized floating-point output Y from the
                   encoded integer input X. Input X can be real or complex values of the
                   following six integer data types: uint8, uint16, uint32, int8, int16, or
                   int32.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.UniformEncoder

Purpose   Quantize and encode floating-point input into integer output

Description   The `UniformEncoder` object quantizes floating-point input, using
                   the precision you specify in the `NumBits` property, and encodes the
                   quantized input into integer output. The operations of the uniform
                   encoder adhere to the definition for uniform encoding specified in ITU-T
                   Recommendation G.701.

                   To quantize and encode a floating-point input into an integer output:

                   1 Define and set up your uniform encoder. See "Construction" on page
                   5-1218.

                   2 Call `step` to encode the input according to the properties of
                   `dsp.UniformEncoder`. The behavior of `step` is specific to each object
                   in the toolbox.

Construction   `H = dsp.UniformEncoder` returns a uniform encoder, `H`, that quantizes
                   floating-point input samples and encodes them as integers using
                   2*N*-level quantization, where *N* is an integer.

                   `H = dsp.UniformEncoder('`*PropertyName*`',`*PropertyValue*`,...)`
                   returns an uniform encoder, `H`, with each specified property set to the
                   specified value.

                   `H = dsp.UniformEncoder(peakvalue,numbits,`
                   `'`*PropertyName*`',`*PropertyValue*`,...)` returns a uniform encoder, `H`,
                   with the `PeakValue` property set to `peakvalue`, the `NumBits` property
                   set to `numbits`, and other specified properties set to the specified values.

Properties   **PeakValue**

                   Largest input amplitude to be encoded

                   Specify the largest input amplitude to be encoded, as a
                   nonnegative numeric scalar. If the real or imaginary input are
                   outside of the interval $[-P, (1 - 2^{(1-B)})P]$, where P is the peak value
                   and B is the value of the `NumBits` property, the uniform encoder

saturates (independently for complex inputs) at those limits. The default is 1.

NumBits

Number of bits needed to represent output

Specify the number of bits needed to represent the integer output as an integer value between 2 and 32. The number of levels at which the uniform encoder quantizes the floating-point input is 2^B, where B is the number of bits. The default is 8.

OutputDataType

Data type of output

Specify the data type of the output as Unsigned integer or Signed integer. Unsigned outputs are uint8, uint16, or uint32, and signed outputs are int8, int16, or int32. The quantized inputs are linearly (uniformly) mapped to the intermediate integers in the interval $[0, 2^{(B-1)}]$ when you set this property to Unsigned integer, and in the interval $[-2^{(B-1)}, 2^{(B-1)} - 1]$ when you set this property to Signed integer. The variable B in both expressions corresponds to the value of the NumBits property.

Methods

clone	Create uniform encoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Quantize and encode input

dsp.UniformEncoder

Examples   Encode a sequence:

```
hue = dsp.UniformEncoder;
hue.PeakValue = 2;
hue.NumBits = 4;
hue.OutputDataType = 'Signed integer';
x = [-1:0.01:1]'; % Create an input sequence
x_encoded = step(hue, x);
plot(x, x_encoded,'.');
xlabel('Input'); ylabel('Encoded Output'); grid
```

Algorithms   This object implements the algorithm, inputs, and outputs described
on the Uniform Encoder block reference page. The object properties
correspond to the block parameters.

See Also   dsp.UniformDecoder

Purpose Create uniform encoder object with same property values

Syntax C = clone(H)

Description C = clone(H) creates a UniformEncoder object C, with the same property values as H. The clone method creates a new unlocked object.

dsp.UniformEncoder.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

dsp.UniformEncoder.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the UniformEncoder object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Quantize and encode input

Syntax   Y = step(H,X)

Description   Y = step(H,X) quantizes and encodes the input X to Y. Input X can
                   be real or complex, double-, or single-precision values. The uniform
                   encoder chooses the output data type according to the following table.

Number of Bits	Unsigned Integer	Signed Integer
2 to 8	uint8	int8
9 to 16	uint16	int16
17 to 32	uint32	int32

The row corresponds to the value of the NumBits property, and the
column corresponds to the value of the OutputDataType property.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

| **Purpose** | Solve upper-triangular matrix equation |

Description   The `UpperTriangularSolver` object solves $UX = B$ for X when U is a square, upper-triangular matrix with the same number of rows as B.

To solve $UX = B$:

1 Define and set up your linear system solver. See "Construction" on page 5-1227.

2 Call `step` to solve the equation according to the properties of `dsp.UpperTriangularSolver`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.UpperTriangularSolver` returns a linear system solver, `H`, used to solve $UX = B$ where U is an upper (or unit-upper) triangular matrix.

`H = dsp.UpperTriangularSolver('`*PropertyName*`',`*PropertyValue*`,...)` returns a linear system solver, `H`, with each specified property set to the specified value.

Properties   **OverwriteDiagonal**

Replace diagonal elements of input with ones

When you set this property to `true`, the linear system solver replaces the elements on the diagonal of the input, *U*, with ones. Set this property to either `true` or `false`. The default is `false`.

RealDiagonalElements

Indicate that diagonal of complex input is real

When you set this property to `true`, the linear system solver optimizes computation speed if the diagonal elements of complex input, *U*, are real. This property applies only when you set the `OverwriteDiagonal` property to `false`. Set this property to either `true` or `false`. The default is `false`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

ProductDataType

Data type of product

Specify the product data type as `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `ProductDataType` property to `Custom`. The default is `numerictype([],32,30)`.

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as `Full precision`, `Same as first input`, `Same as product`, or `Custom`. The default is `Full precision`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,30)`.

OutputDataType

Data type of output

Specify the output data type as `Same as first input` or `Custom`. The default is `Same as first input`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],16,15)`.

Methods

clone	Create upper triangular solver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Solve matrix equation for specified inputs

Examples

Solve an upper-triangular matrix equation:

```
huptriang = dsp.UpperTriangularSolver;
```

dsp.UpperTriangularSolver

```
u = triu(rand(4, 4));
b = rand(4, 1);
% Check that result is the solution to the linear
% equations.
x1 = inv(u)*b
x = step(huptriang, u, b)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on
the Backward Substitution block reference page. The object properties
correspond to the block parameters.

See Also   dsp.LowerTriangularSolver

Purpose   Create upper triangular solver object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates an UpperTriangularSolver object C, with the
                   same property values as H. The clone method creates a new unlocked
                   object.

dsp.UpperTriangularSolver.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                     method.

                     The getNumOutputs method returns a positive integer that is the
                     number of outputs from the step method. This value will change if you
                     alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the UpperTriangularSolver object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Solve matrix equation for specified inputs

Syntax   X = step(H,U,B)

Description   X = step(H,U,B) computes the solution, *X*, of the matrix equation
                    UX = B, where *U* is a square, upper-triangular matrix with the same
                    number of rows as the matrix *B*.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

Purpose   Delay input by time-varying fractional number of sample periods

Description The `VariableFractionalDelay` object delays the input by a
                 time-varying fractional number of sample periods.

                 To delay the input by a time-varying fractional number of sample
                 periods:

                 1 Define and set up your variable fractional delay object. See
                       "Construction" on page 5-1237.

                 2 Call `step` to delay the input according to the properties of
                       `dsp.VariableFractionalDelay`. The behavior of `step` is specific to
                       each object in the toolbox.

Construction H = `dsp.VariableFractionalDelay` returns a variable fractional delay
                 System object, `H`, that delays a discrete-time input by a time-varying
                 fractional number of sample periods.

                 `H =`
                 `dsp.VariableFractionalDelay('`*PropertyName*`',`*PropertyValue*`,...)`
                 returns a variable fractional delay System object, `H`, with each
                 property set to the specified value.

Properties   **InterpolationMethod**

                 Interpolation method

                 Specify the method by which the block interpolates between
                 adjacent stored samples to obtain a value for the sample indexed
                 by the input. You can set this property to one of | `Linear` | `FIR` |
                 `Farrow` |. When you set this property to `FIR`, the object uses the
                 Signal Processing Toolbox `intfilt` function to compute an FIR
                 filter for interpolation. The default is `Linear`.

                 FilterHalfLength

                 FIR interpolation filter half-length

Specify the half-length of the FIR interpolation filter as a positive scalar integer. This property applies only when you set the `InterpolationMethod` property to `FIR`. For periodic signals, a larger value of this property (that is, a higher order filter) yields a better estimate of the delayed output sample. Setting this property to a value between `4` and `6` (that is, a 7th to 11th order filter) is usually adequate. The default is `4`.

FilterLength

Length of Farrow filter

Specify the length of the FIR filter implemented using the Farrow structure, as a positive scalar integer. This property applies only when you set the `InterpolationMethod` property to `Farrow`. The default is `4`.

InterpolationPointsPerSample

Number of interpolation points per input sample

Specify the number of interpolation points per input sample at which a unique FIR interpolation filter is computed. You must specify the number of interpolation points per input sample as a positive scalar integer. This property applies only when you set the `InterpolationMethod` property to `FIR`. The default is `10`.

Bandwidth

Normalized input bandwidth

Specify the bandwidth to which you want to constrain the interpolated output samples. You must enter the bandwidth as a scalar value between `0` and `1`. You can use this property to take advantage of the bandlimited frequency content of the input. For example, if the input signal does not have frequency content above `Fs/4` (where `Fs` is the sampling frequency), you can specify a value of `0.5` for the `Bandwidth` property. A value of `1` for the `Bandwidth` property corresponds to half the sampling frequency (`Fs`). This property applies only when you set the `InterpolationMethod`property to `FIR`. The default is `1`.

InitialConditions

Initial values in memory

Specify the values with which the object's memory is initialized. The dimensions of this property can vary depending on the setting of the `FrameBasedProcessing` property, and whether you want fixed or time-varying initial conditions. The default is `0`.

When you set the `FrameBasedProcessing` property to `false`, the object supports N-D input arrays. For an M-by-N-by-P sample-based input array `U`, you can set the `InitialConditions` property as follows:

- To specify fixed initial conditions, set the `InitialConditions` property to a scalar value. The object initializes every sample of every channel in memory using the value you specify.

- The dimensions you specify for time-varying initial conditions depend on the value of the `InterpolationMethod` property:

  - If you set the `InterpolationMethod` property to `Linear`, set the `InitialConditions` property to an array of dimension Mx-by-N-by-P-by-D. The object uses the values in this array to initialize memory samples `U(2:D+1)`, where `D` is the value of the `MaximumDelay` property.

  - If you set the `InterpolationMethod` property to either `FIR` or `Farrow`, set the `InitialConditions` property to an array of dimension M-by-N-by-P-by-(D+L). The object uses the values in this array to initialize memory samples `U(2:D+1)`, where `D` is the value of the `MaximumDelay` property. For FIR interpolation, `L` is the value of the `FilterHalfLength` property. For Farrow interpolation, `L` equals `floor` of half the value of the `FilterLength` property (`floor(FilterLength/2)`).

When you set the `FrameBasedProcessing` property to `true`, the object treats each of the N input columns as a frame containing M sequential time samples from an independent channel.

For an M-by-N frame-based input matrix U, you can set the `InitialConditions` property as follows:

- To specify fixed initial conditions, set the `InitialConditions` property to a scalar value. The object initializes every sample of every channel in memory using the value you specify.

- The dimensions you specify for time-varying initial conditions depend on the value of the `InterpolationMethod` property. To specify different time-varying initial conditions for each channel, set the `InitialConditions` property as follows:

  - If you set the `InterpolationMethod` property to `Linear`, set the `InitialConditions` property to an array of size 1-by-N-by-D, where D is the value of the `MaximumDelay` property.

  - If you set the `InterpolationMethod` property to `FIR` or `Farrow`, set the `InitialConditions` property to an array of size 1-by-N-by-(D+L), where D is the value of the `MaximumDelay` property. For FIR interpolation, L is the value of the `FilterHalfLength` property. For Farrow interpolation, L equals `floor` of half the value of the `FilterLength` property (`floor(FilterLength/2)`).

MaximumDelay

Maximum delay

Specify the maximum delay the object can produce for any sample. The maximum delay must be a positive scalar integer value. The object clips input delay values greater than the `MaximumDelay` to the `MaximumDelay`. The default is `100`.

DirectFeedthrough

Allow direct feedthrough

When you set this property to `true`, the object allows direct feedthrough. When you set this property to `false`, the object increases the minimum possible delay by one. The default is `true`.

FIRSmallDelayAction

Action for small input delay values in FIR interpolation mode

Specify the action the object should take for small input delay values when using the FIR interpolation method. You can set this property to `Clip to the minimum value necessary for centered kernel`, or `Switch to linear interpolation if kernel cannot be centered`. This property applies only when you set the `InterpolationMethod` property to `FIR`. The default is `Clip to the minimum value necessary for centered kernel`.

FarrowSmallDelayAction

Action for small input delay values in Farrow interpolation mode

Specify the action the object should take for small input delay values when using the Farrow interpolation method. You can set this property to `Clip to the minimum value necessary for centered kernel`, or `Use off-centered kernel`. This property applies only when you set the `InterpolationMethod` property to `Farrow`. The default is `Clip to the minimum value necessary for centered kernel`.

FrameBasedProcessing

Treat input as frame based or sample based

Set this property to `true` to enable frame-based processing. When you do so, the object accepts M-by-N input matrices and treats each of the `N` input columns as a frame containing `M` sequential time samples from an independent channel. Set this property to `false` to enable sample-based processing. When you do so, the object supports N-D inputs and treats each element of the input as a separate channel. The default is `true`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | `Ceiling` | `Convergent`, | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero` |. The default is `Zero`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | `Wrap` | `Saturate` |. The default is `Wrap`.

CoefficientsDataType

Coefficient word and fraction lengths

Specify the coefficients data type as one of | `Same word length as input` | `Custom` |. The default is `Same word length as input`.

CustomCoefficientsDataType

Coefficients word length

Specify the coefficients word length as an unscaled `numerictype` object with a `Signedness` of `Auto`. If the `InterpolationMethod` is `Linear`, specify that the `numerictype` object is unsigned; otherwise the object should be signed. This property applies only when you set the `CoefficientsDataType` property to `Custom`. The default is `numerictype([],32)`.

ProductPolynomialValueDataType

Product polynomial values word and fraction lengths

Specify the product polynomial value data type as one of | `Same as first input` | `Custom` |. This property applies only when you set the `InterpolationMethod` property to `Farrow`. The default is `Same as first input`.

CustomProductPolynomialValueDataType

Product polynomial values word and

fraction lengths Specify the product polynomial values fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `InterpolationMethod` property to `Farrow` and the `ProductPolynomialValueDataType` property to `Custom`. The default is `numerictype([],32,10)`.

AccumulatorPolynomialValueDataType

Accumulator polynomial value word and fraction lengths

Specify the accumulator polynomial value data type as one of | `Same as first input` | `Custom` |. This property applies only when you set the `InterpolationMethod` property to `Farrow`. The default is `Same as first input`.

CustomAccumulatorPolynomialValueDataType

Accumulator polynomial value word and fraction lengths

Specify the data type of the accumulator polynomial values as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `InterpolationMethod` property to `Farrow` and the `AccumulatorPolynomialValueDataType` property to `Custom`. The default is `numerictype([],32,10)`.

MultiplicandPolynomialValueDataType

Multiplicand polynomial value word and fraction lengths

Specify the multiplicand polynomial values data type as one of | `Same as first input` | `Custom` |. This property applies only when you set the `InterpolationMethod` property to `Farrow`. The default is `Same as first input`.

CustomMultiplicandPolynomialValueDataType

Multiplicand polynomial value word and fraction lengths

Specify the fixed-point data type of the multiplicand polynomial values as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `InterpolationMethod` property to `Farrow` and the

MultiplicandPolynomialValueDataType property to `Custom`.
The default is `numerictype([],32,10)`.

ProductDataType

Product word and fraction lengths

Specify the product data type as one of | `Same as first input` |
`Custom` |. The default is `Same as first input`.

CustomProductDataType

Product word and fraction lengths

Specify the product data type as a scaled `numerictype` object
with a `Signedness` of `Auto`. This property applies only when
you set the `ProductDataType` property to `Custom`. The default is
`numerictype([],32,10)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator data type as one of | `Same as product`
| `Same as first input` | `Custom` |. The default is `Same as`
`product`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the fixed-point accumulator data type as a scaled
`numerictype` object with a `Signedness` of `Auto`. This property
applies only when you set the `AccumulatorDataType` property to
`Custom`. The default is `numerictype([],32,10)`.

OutputDataType

Output word and fraction lengths

Specify the output data type as one of | `Same as accumulator`
| `Same as first input` | `Custom` |. The default is `Same as`
`accumulator`.

CustomOutputDataType

Output word and fraction lengths

Specify the data type of the output as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies only when you set the `OutputDataType` property to `Custom`. The default is `numerictype([],32,10)`.

Methods

clone	Create variable fractional delay object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
info	Characteristic information about valid delay range
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of variable fractional delay object
step	Delay input by time-varying fractional number of sample periods

Examples

Delay a signal by a varying fractional number of sample periods:

```
hsr = dsp.SignalSource;
hvfd = dsp.VariableFractionalDelay;
hLog = dsp.SignalSink;

for ii = 1:10
    delayedsig = step(hvfd, step(hsr), ii/10);
     step(hLog, delayedsig);
```

dsp.VariableFractionalDelay

```
end

sigd = hLog.Buffer;

% The output sigd corresponds to the values of the delayed signal
% that are sampled at fixed-time intervals. For visualization
% purposes, we can instead plot the time instants at which the
% amplitudes of signal samples are constant by treating the
% signals as the sampling instants.

stem(hsr.Signal, 1:10, 'b')
hold on;
stem(sigd.', 1:10, 'r');
legend('Original signal',...
    'Variable fractional delayed signal', ...
    'Location','best')
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Variable Fractional Delay block reference page. The object properties correspond to the block properties, except:

When you set the `DirectFeedthrough` property of the System object to `true`, the object allows direct feedthrough. This behavior is different from the way the block behaves when you select the corresponding **Disable direct feedthrough by increasing minimum possible delay by one** check box on the block dialog. When you enable this block parameter, the block does not allow direct feedthrough.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the `FrameBasedProcessing` property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.VariableIntegerDelay | dsp.Delay | dsp.DelayLine

Purpose   Create variable fractional delay object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a VariableFractionalDelay System object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

dsp.VariableFractionalDelay.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                  the step method.

                  The getNumInputs method returns a positive integer that is the number
                  of expected inputs (not counting the object itself) to the step method.
                  This value will change if you alter any properties that turn inputs on or
                  off. You must call the step method with the number of input arguments
                  equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
method.

The getNumOutputs method returns a positive integer that is the
number of outputs from the step method. This value will change if you
alter any properties that turn outputs on or off.

dsp.VariableFractionalDelay.info

Purpose   Characteristic information about valid delay range

Syntax   `S = info(H)`

Description   `S = info(H)` returns a structure, `S`, containing characteristic information about the `VariableFractionalDelay` System object, `H`. `S` has one field, `ValidDelayRange` which is a string containing the possible range of delay values based on the current property values of the `VariableFractionalDelay` object. The `ValidDelayRange` is in the format `[MinValidDelay, MaxValidDelay]`. The object clips all input delay values to be within this `ValidDelayRange`.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the VariableFractionalDelay
System object.

The isLocked method returns a logical value that indicates whether
input attributes and nontunable properties for the object are locked. The
object performs an internal initialization the first time the step method
is executed. This initialization locks nontunable properties and input
specifications, such as dimensions, complexity, and data type of the
input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Reset internal states of variable fractional delay object

Syntax   reset(H)

Description   reset(H) resets the internal states of the VariableFractionalDelay
                  System object H to their initial values.

dsp.VariableFractionalDelay.step

Purpose   Delay input by time-varying fractional number of sample periods

Syntax   Y = step(H,X,D)

Description   Y = step(H,X,D) delays the input X by D samples, where D should be less than or equal to the value specified in the MaximumDelay property.

> **Note** The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

Purpose	Delay input by time-varying integer number of sample periods
Description	The `VariableIntegerDelay` object delays input by time-varying integer number of sample periods.

To delay the input by a time-varying integer number of sample periods:

1 Define and set up your variable integer delay object. See "Construction" on page 5-1255.

2 Call `step` to delay the input according to the properties of `dsp.VariableIntegerDelay`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.VariableIntegerDelay` returns a variable integer delay System object, `H`, that delays discrete-time input by a time-varying integer number of sample periods.

```
H =
dsp.VariableIntegerDelay('PropertyName',PropertyValue,...)
```
returns a variable integer delay System object, `H`, with each property set to the specified value.

Properties   **MaximumDelay**

Maximum delay

Specify the maximum delay the object can produce for any sample. The maximum delay must be a positive scalar integer value. The object clips input delay values greater than the `MaximumDelay` to the `MaximumDelay`. The default is `100`.

InitialConditions

Initial values in memory

Specify the values with which the object's memory is initialized. The dimensions of this property can vary depending on the setting of the `FrameBasedProcessing` property, and whether you want fixed or time-varying initial conditions. The default is `0`.

When you set the `FrameBasedProcessing` property to `false`, the object supports N-D input arrays. For an M-by-N-by-P sample-based input array U, you can set the `InitialConditions` property as follows:

- To specify fixed initial conditions, set the `InitialConditions` property to a scalar value. The object initializes every sample of every channel in memory using the value you specify.

- To specify time-varying initial conditions, set the `InitialConditions` property to an array of dimension M-by-N-by-P-by-D. The object uses the values in this array to initialize memory samples `U(2:D+1)`, where D is the value of the `MaximumDelay` property.

When you set the `FrameBasedProcessing` property to `true`, the object treats each of the N input columns as a frame containing M sequential time samples from an independent channel. For an M-by-N frame-based input matrix U, you can set the `InitialConditions` property as follows:

- To specify fixed initial conditions, set the `InitialConditions` property to a scalar value. The object initializes every sample of every channel in memory using the value you specify.

- To specify different time-varying initial conditions for each channel, set the `InitialConditions` property to an array of size 1-by-N-by-D, where D is the value of the `MaximumDelay` property.

DirectFeedthrough

Allow direct feedthrough

When you set this property to `true`, the object allows direct feedthrough. When you set this property to `false`, the object increases the minimum possible delay by one. The default is `true`.

FrameBasedProcessing

Treat input as frame based or sample based

Set this property to `true` to enable frame-based processing. When you do so, the object accepts M-by-N input matrices and treats each of the N input columns as a frame containing M sequential time samples from an independent channel. Set this property to `false` to enable sample-based processing. When you do so, the object supports N-D inputs and treats each element of the input as a separate channel. The default is `true`.

Methods

clone	Create variable integer delay object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of variable integer delay object
step	Delay input by time-varying integer number of sample periods

Examples

Delay a signal by a varying integer number of sample periods:

```
h = dsp.VariableIntegerDelay;
x = 1:100;
ii = 0;
k = 0;
yout = [];

while(ii+10 <= 100)
```

```
            y = step(h, x(ii+1:ii+10)',k*ones(10,1));
            yout = [yout;y];
            ii = ii+10;
            k = k+1;
end

stem(x,'b');
hold on; stem(yout,'r');
legend('Original Signal', 'Variable Integer Delayed Signal')
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Variable Integer Delay block reference page. The object properties correspond to the block properties, except:

> When you set the `DirectFeedthrough` property of the System object to `true`, the object allows direct feedthrough. This behavior is different from the way the block behaves when you select the corresponding **Disable direct feedthrough by increasing minimum possible delay by one** check box on the block dialog. When you enable this block parameter, the block does not allow direct feedthrough.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the `FrameBasedProcessing` property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   dsp.VariableFractionalDelay | dsp.Delay | dsp.DelayLine

Purpose   Create variable integer delay object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a VariableIntegerDelay System object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to the step method.

The getNumInputs method returns a positive integer that is the number of expected inputs (not counting the object itself) to the step method. This value will change if you alter any properties that turn inputs on or off. You must call the step method with the number of input arguments equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

dsp.VariableIntegerDelay.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the `VariableIntegerDelay`
                System object.

                The `isLocked` method returns a logical value that indicates whether
                input attributes and nontunable properties for the object are locked. The
                object performs an internal initialization the first time the `step` method
                is executed. This initialization locks nontunable properties and input
                specifications, such as dimensions, complexity, and data type of the
                input data. After locking, the `isLocked` method returns a `true` value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.VariableIntegerDelay.reset

Purpose   Reset internal states of variable integer delay object

Syntax   reset(H)

Description   reset(H) resets the internal states of the `VariableIntegerDelay` System object `H` to their initial values.

Purpose   Delay input by time-varying integer number of sample periods

Syntax   Y = step(H,X,D)

Description   Y = step(H,X,D) delays the input X by D samples, where D should be
less than or equal to the value specified in the MaximumDelay property
and greater than or equal to 0. The object clips delay values greater
than the MaximumDelay to the MaximumDelay, and clips values less than
zero to zero. If you enter a noninteger delay value, the object rounds
that value to the nearest integer value.

Note The object performs an initialization the first time the step
method is executed. This initialization locks nontunable properties and
input specifications, such as dimensions, complexity, and data type
of the input data. If you change a nontunable property or an input
specification, the System object issues an error. To change nontunable
properties or inputs, you must first call the release method to unlock
the object.

---

dsp.Variance

Purpose	Variance of input or sequence of inputs
Description	The Variance object computes variance for an input or sequence of inputs.

To compute the variance of an input or sequence of inputs:

1 Define and set up your variance System object. See "Construction" on page 5-1266.

2 Call step to compute the variance according to the properties of dsp.Variance. The behavior of step is specific to each object in the toolbox.

Construction   H = dsp.Variance returns a variance System object, H, that computes the variance of an input or a sequence of inputs over the specified Dimension.

H = dsp.Variance('*PropertyName*',*PropertyValue*,...) returns a variance System object, H, with each specified property set to the specified value.

Properties   **RunningVariance**

Enable calculation over time

Set this property to true to enable variance calculation over successive calls to the step method. The default is false.

ResetInputPort

Enable reset input port

Set this property to true to enable reset input port. When you set the property to true, specify a reset input for the step method. The running variance resets anytime the variance object achieves the condition you specify for the ResetCondition property. This property applies when you set the RunningVariance property to true. The default is false.

ResetCondition

Reset condition for running variance mode

Specify which event resets the running variance as one of | Rising edge | Falling edge | Either edge | Non-zero |. This property applies when you set the ResetInputPort property to true.

Dimension

Dimension to operate along

Specify how the object performs the variance calculation over the data as one of | All | Row | Column | Custom |. This property applies when you set the RunningVariance property to false. The default is Column.

CustomDimension

Numerical dimension to operate along

Specify the input signal dimension (one-based value) the object uses to compute variance. The cannot exceed the number of dimensions in the input signal. This property applies when you set the Dimension property to Custom. The default is 1.

ROIProcessing

Enable region-of-interest processing

Set this property to true to enable calculating the variance within a particular region of each image. This property applies when you set the RunningVariance property to false and the Dimension property to All. The default is false. Full ROI processing support requires a Computer Vision System Toolbox license. With only the DSP System Toolbox license, Rectangles is the only selection for the ROIForm property.

ROIForm

Define the type of region of interest.

Specify the type of region of interest as one of | Rectangles | Lines | Label matrix | Binary mask |. This property applies

when you set the `ROIProcessing` property to `true`. The default is `Rectangle`.

ROIPortion

Calculate over entire ROI or just perimeter

Specify the region over which to calculate variance as one of | `Entire ROI` | `ROI perimeter` |. This property applies when you set the `ROIForm` property to `Rectangles`. The default is `Entire ROI`.

ROIStatistics

Statistics for each ROI or one for all ROIs

Specify if statistics calculations are one of | `Individual statistics for each ROI` | `Single statistic for all ROIs` |. This property applies when `ROIForm` property is not `Binary mask`. The default is `Individual statistics for each ROI`.

ValidityOutputPort

Enable output of validity check of ROI or label numbers

Indicate whether to return the validity of the specified ROI being completely inside image when the ROIForm property is `Lines` or `Rectangles`. Indicate whether to return the validity of the specified label numbers when the ROIForm property is `Label Matrix`. This property applies when you set the ROIForm property to anything except `Binary Mask`. The default is `false`.

FrameBasedProcessing

Enable frame-based processing

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. This property applies when you set the `RunningVariance` property to `true`. The default is `true`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of | Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero |. The default is Floor

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of | Wrap | Saturate |. The default is Wrap.

InputSquaredProductDataType

Input-squared product word and fraction lengths

Specify the input-squared product fixed-point data type as one of | Same as input | Custom |. The default is Same as input.

CustomInputSquaredProductDataType

Input-squared product word and fraction lengths

Specify the input-squared product fixed-point type as a scaled numerictype object with a Signedness of Auto. This property applies when you set the InputSquaredProductDataType property to Custom. The default is numerictype([],32,15).

InputSumSquaredProductDataType

Input-sum-squared product word and fraction lengths

Specify the input-sum-squared product fixed-point data type as one of | Same as input-squared product | Custom |. The default is Same as input-squared product.

CustomInputSumSquaredProductDataType

Input-sum-squared product word and fraction lengths

Specify the input-sum-squared product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `InputSumSquaredProductDataType` property to `Custom`. The default is `numerictype([],32,23)`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as one of | `Same as input-squared product` | `Same as input` | `Custom` |. The default is `Same as input-squared product`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`. The default is `numerictype([],32,0)`.

OutputDataType

Output word and fraction lengths

Specify the output fixed-point data type as one of | `Same as input-squared product` | `Same as input` | `Custom` |. The default is `Same as input-squared product`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property only applies when the `OutputDataType` property to `Custom`. The default is `numerictype([],16,0)`.

Methods	clone	Create variance object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	reset	Reset variance to zero
	step	Variance of input

Examples Compute the running variance for a signal:

```
hvar = dsp.Variance;
hvar.RunningVariance = true;
x = randn(100,1);
y = step(hvar, x);
% y(i) is the running variance of all values in the vector x(1:i)
```

Algorithms This object implements the algorithm, inputs, and outputs described on the Variance block reference page. The object properties correspond to the block parameters, except:

- **Treat sample-based row input as a column** block parameter is not supported by the dsp.Variance object.

- **Reset port** block parameter corresponds to both the ResetCondition and the ResetInputPort object properties.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object

dsp.Variance

uses the `FrameBasedProcessing` property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

See Also   `dsp.Mean` | `dsp.RMS` | `dsp.StandardDeviation`

Purpose   Create variance object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a Variance System object C, with the same
                   property values as H. The clone method creates a new unlocked object
                   with uninitialized states.

dsp.Variance.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                     the step method.

                     The getNumInputs method returns a positive integer that is the number
                     of expected inputs (not counting the object itself) to the step method.
                     This value will change if you alter any properties that turn inputs on or
                     off. You must call the step method with the number of input arguments
                     equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                    method.

                    The getNumOutputs method returns a positive integer that is the
                    number of outputs from the step method. This value will change if you
                    alter any properties that turn outputs on or off.

dsp.Variance.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the Variance System object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.Variance.reset

Purpose   Reset variance to zero

Syntax   reset(H)

Description   reset(H) sets the variance to zero when the RunningVariance property
                    is true.

Purpose   Variance of input

Syntax   `Y = step(H,X)`
`Y = step(H,X,R)`
`VAR2D = step(H,X,ROI)`
`VAR2D = step(H,X,LABEL,LABELNUMBERS)`
`[VAR2D,FLAG] = step(H,X,ROI)`
`[VAR2D,FLAG] = step(H,X,LABEL,LABELNUMBERS)`

Description   `Y = step(H,X)` computes the variance, `Y`, of input `X` over successive calls to the `step` method, when the `RunningVariance` property is `true`.

`Y = step(H,X,R)` resets its state based on the value of reset signal `R`, the `ResetInputPort` property and the `ResetCondition` property. This option applies when the `RunningVariance` property is `true` and the `ResetInputPort` property is set to `true`.

`VAR2D = step(H,X,ROI)` computes the variance of input image, `X`, within the given region of interest, `ROI`, when the `ROIProcessing` property is `true` and the `ROIForm` property is `Lines`, `Rectangles` or `Binary mask`. Full `ROI` processing support requires a Computer Vision System Toolbox license. With only the DSP System Toolbox license, the `ROIForm` property only supports `Rectangles`.

`VAR2D = step(H,X,LABEL,LABELNUMBERS)` computes the variance of input image, `X`, for region labels contained in vector `LABELNUMBERS`, with matrix `LABEL` marking pixels of different regions. This option is available when the `ROIProcessing` property is `true` and the `ROIForm` property is `Label matrix`.

`[VAR2D,FLAG] = step(H,X,ROI)` returns `FLAG` which indicates whether the given region of interest is within the image bounds when both the `ROIProcessing` and `ValidityOutputPort` properties are `true` and the `ROIForm` property is set to `Lines`, `Rectangles` or `Binary mask`.

`[VAR2D,FLAG] = step(H,X,LABEL,LABELNUMBERS)` returns `FLAG` which indicates whether the input label numbers are valid when both the `ROIProcessing` and `ValidityOutputPort` properties are `true` and the `ROIForm` property is set to `Label matrix`.

> **Note** The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Purpose   Vector quantizer codeword for given index value

Description   The `VectorQuantizerDecoder` object returns the vector quantizer
                     codeword for a given index value. Each column of the `Codebook` property
                     is a codeword.

                     To obtain the vector quantizer codeword for a given index value:

                     1 Define and set up your vector quantizer decoder. See "Construction"
                        on page 5-1281.

                     2 Call `step` to get the vector quantizer codeword according to the
                        properties of `dsp.VectorQuantizerDecoder`. The behavior of `step` is
                        specific to each object in the toolbox.

Construction   `H = dsp.VectorQuantizerDecoder` creates a vector quantizer
                     decoder System object, `H`, that returns a vector quantizer codeword
                     corresponding to a given, zero-based index value.

                     `H = dsp.VectorQuantizerDecoder('PropertyName',PropertyValue,...)`
                     returns a vector quantizer decoder, `H`, with each specified property set
                     to the specified value.

Properties   **CodebookSource**

                     Source of codebook values

                     Specify the codebook source as `Property` or `Input port`. When
                     you select `Property`, the object reads the codebook from the
                     `codebook` property. When you select `Input port`, the object reads
                     the codebook from the input of the `step` method.

                     The default is `Property`.

                     Codebook

                     codebook

Specify quantized output values as a *k*-by-*N* matrix, where $k \geq 1$ and $N \geq 1$. Each column of the codebook matrix is a codeword, and each codeword corresponds to an index value. This property applies when you set the `CodebookSource` property to `Property`. The default is:

$$\begin{bmatrix} 1.5 \ 13.3 \ 136.4 \ 6.8 \\ 2.5 \ 14.3 \ 137.4 \ 7.8 \\ 3.5 \ 15.3 \ 138.4 \ 8.8 \end{bmatrix}$$

This property is tunable.

OutputDataType

Data type of codebook and quantized output

Specify the data type of the codebook and quantized output values as: `Same as input`, `double`, `single` or `Custom`. This property applies only when you set `CodebookSource` to `Property`. The default is `double`.

Fixed-Point Properties

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a signed or unsigned `numerictype` object. This property applies only when you set the `OutputDataType` property to `Custom`. The default is `numerictype(true,16)`.

Methods		
	clone	Create vector quantizer decoder object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Quantized output values corresponding to input indices

Examples   Given index values as an input, determine the corresponding vector quantized codewords for a specified codebook:

```
hvqdec = dsp.VectorQuantizerDecoder;
hvqdec.Codebook = [1 10 100;2 20 200;3 30 300];
indices = uint8([1 0 2 0]);
qout = step(hvqdec, indices)
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Vector Quantizer Decoder block reference page. The object properties correspond to the block parameters, except:

There is no object property that directly corresponds to the **Action for out of range index value** block parameter. The object sets any index values less than 0 to 0 and any index values greater than or equal to *N* to *N*-1.

See Also   dsp.VectorQuantizerEncoder | dsp.ScalarQuantizerDecoder

Purpose   Create vector quantizer decoder object with same property values

Syntax

Description   C = clone(H) creates a VectorQuantizerDecoder object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
the step method.

The getNumInputs method returns a positive integer that is the number
of expected inputs (not counting the object itself) to the step method.
This value will change if you alter any properties that turn inputs on or
off. You must call the step method with the number of input arguments
equal to the result of getNumInputs(*H*).

dsp.VectorQuantizerDecoder.getNumOutputs

Purpose Number of outputs of step method

Syntax N = getNumOutputs(H)

Description N = getNumOutputs(H) returns the number of outputs, N, of the `step` method.

The `getNumOutputs` method returns a positive integer that is the number of outputs from the `step` method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description   isLocked(H) returns the locked state of the VectorQuantizerDecoder
object H.

The isLocked method returns a logical value that indicates whether
input attributes and nontunable properties for the object are locked. The
object performs an internal initialization the first time the step method
is executed. This initialization locks nontunable properties and input
specifications, such as dimensions, complexity, and data type of the
input data. After locking, the isLocked method returns a true value.

dsp.VectorQuantizerDecoder.release

Purpose   Allow property value and input characteristics changes

Syntax   `release(H)`

Description   `release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note   You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Quantized output values corresponding to input indices

Syntax   Q = step(H,I)
                   Q = step(H,I,C)

Description   Q = step(H,I) returns the quantized output values Q corresponding to
                   the input indices I. The data type of I can be uint8, uint16, uint32,
                   int8, int16, or int32. The OutputDataType property determines the
                   data type of Q.

                   Q = step(H,I,C) uses input C as the codebook values when the
                   CodebookSource property is Input port. The data type of C can be
                   double, single, or fixed-point. The output Q has the same data type
                   as the codebook input C.

                   Note The object performs an initialization the first time the step
                   method is executed. This initialization locks nontunable properties and
                   input specifications, such as dimensions, complexity, and data type
                   of the input data. If you change a nontunable property or an input
                   specification, the System object issues an error. To change nontunable
                   properties or inputs, you must first call the release method to unlock
                   the object.

                   ---

dsp.VectorQuantizerEncoder

Purpose   Vector quantization encoding

Description   The `VectorQuantizerEncoder` object performs vector quantization
                     encoding. The object finds the nearest codeword by computing a
                     distortion based on Euclidean or weighted Euclidean distance.

                     To perform vector quantization encoding:

                     1 Define and set up your vector quantizer encoder. See "Construction"
                        on page 5-1290.

                     2 Call `step` to perform the quantization encoding according to the
                        properties of `dsp.VectorQuantizerEncoder`. The behavior of `step` is
                        specific to each object in the toolbox.

Construction   `H = dsp.VectorQuantizerEncoder` returns a vector quantizer encoder
                     System object, `H`. This object finds a zero-based index of the nearest
                     codeword for each given input column vector.

                     `H =`
                     `dsp.VectorQuantizerEncoder('`*PropertyName*`',`*PropertyValue*`,...)`
                     returns a vector quantizer encoder System object, `H`, with each specified
                     property set to the specified value.

Properties   **CodebookSource**

                        Source of codebook values

                        Specify how to determine the codebook values as `Property` or
                        `Input port`. The default is `Property`.

                     Codebook

                        Codebook

                        Specify the codebook to which the input column vector or matrix
                        is compared, as a k-by-N matrix. Each column of the codebook
                        matrix is a codeword, and each codeword corresponds to an index
                        value. The codeword vectors must have the same number of rows
                        as the input. The first codeword vector corresponds to an index

value of 0, the second codeword vector corresponds to an index value of 1, and so on. This property applies when you set the CodebookSource property to Property. The default is:

$$\begin{bmatrix} 1.5 & 13.3 & 136.4 & 6.8 \\ 2.5 & 14.3 & 137.4 & 7.8 \\ 3.5 & 15.3 & 138.4 & 8.8 \end{bmatrix}$$

This property is tunable.

DistortionMeasure

Distortion calculation method

Specify how to calculate the distortion as Squared error or Weighted squared error. If you set this property to Squared error, the object calculates the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. If you set this property to Weighted squared error, the object calculates the distortion by evaluating a weighted Euclidean distance using a weighting factor to emphasize or deemphasize certain input values. The default is Squared error.

WeightsSource

Source of weighting factor

Specify how to determine weighting factor as Property or Input port. This property applies when you set the DistortionMeasure property to Weighted squared error. The default is Property.

Weights

Weighting factor

Specify the weighting factor as a vector of length equal to the number of rows of the input. This property applies when you set the DistortionMeasure property to Weighted squared error and WeightsSource property is Property. The default is [1 1 1]. This property is tunable.

TiebreakerRule

Behavior when input column vector is equidistant from two codewords.

Specify whether to represent the input column vector by the lower index valued codeword or higher indexed valued codeword when an input column vector is equidistant from two codewords. You can set this property to `Choose the lower index` or `Choose the higher index`. The default is `Choose the lower index`.

CodewordOutputPort

Enable output of codeword value

Set this property to `true` to output the codeword vectors nearest to the input column vectors. The default is `false`.

QuantizationErrorOutputPort

Enable output of quantization error

Set this property to `true` to output the quantization error value that results when the object represents the input column vector by the nearest codeword. The default is `false`.

OutputIndexDataType

Data type of index output

Specify the data type of the index output as: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`. The default is `int32`.

Fixed-Point Properties

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as `Ceiling`, `Convergent`, `Floor`, `Nearest`, `Round`, `Simplest` or `Zero`. The default is `Floor`.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as `Wrap` or `Saturate`. The default is `Wrap`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as `Same as input` or `Custom`. The default is `Same as input`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`.

AccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point data type as `Same as input`, or `Custom`. The default is `Same as product`.

CustomAccumulatorDataType

Accumulator word and fraction lengths

Specify the accumulator fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `AccumulatorDataType` property to `Custom`.

Methods

clone	Create vector quantizer encoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method

dsp.VectorQuantizerEncoder

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Quantization region to which input belongs

Examples

Find the indices of nearest codewords based on Euclidean distances:

```
hvqenc = dsp.VectorQuantizerEncoder(...
'Codebook', [-1 -1 1 1;1 -1 -1 1], ...
'CodewordOutputPort', true, ...
'QuantizationErrorOutputPort', true, ...
'OutputIndexDataType', 'uint8');
% Generate an input signal with some additive noise
x = sign(rand(2,40)-0.5) + 0.1*randn(2,40);
[ind, cw, err] = step(hvqenc, x);
plot(cw(1,:), cw(2,:), 'rO', x(1,:), x(2,:), 'g.');
legend('Quantized', 'Inputs', 'location', 'best');
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Vector Quantizer Encoder block reference page. The object properties correspond to the block parameters.

See Also

dsp.VectorQuantizerDecoder | dsp.ScalarQuantizerEncoder

Purpose   Create vector quantizer encoder object with same property values

Syntax

Description   `C = clone(H)` creates a `VectorQuantizerEncoder` object `C`, with the same property values as `H`. The clone method creates a new unlocked object.

dsp.VectorQuantizerEncoder.getNumInputs

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description N = getNumInputs(H) returns the number of expected inputs, N, to
                 the step method

                 The getNumInputs method returns a positive integer that is the number
                 of expected inputs (not counting the object itself) to the step method.
                 This value will change if you alter any properties that turn inputs on or
                 off. You must call the step method with the number of input arguments
                 equal to the result of getNumInputs(*H*).

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step
                   method

                   The getNumOutputs method returns a positive integer that is the
                   number of outputs from the step method. This value will change if you
                   alter any properties that turn outputs on or off.

dsp.VectorQuantizerEncoder.isLocked

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the VectorQuantizerEncoder System object H.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description   release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

---

dsp.VectorQuantizerEncoder.step

Purpose   Quantization region to which input belongs

Syntax
```
INDEX = step(H,INPUT)
INDEX = step(H,...,CODEBOOK)
INDEX = step(H,...,WEIGHTS)
[..., CODEWORD] = step(H, ...)
[..., QERR] = step(H, ...)
```

Description   `INDEX = step(H,INPUT)` returns `INDEX`, a scalar or column vector representing the quantization region(s) to which `INPUT` belongs. `INPUT` can be a column vector of size k-by-1 or an M multichannel matrix of dimensions k-by-M, where k is the length of each codeword in the codebook. All inputs to the object can be real floating-point or fixed-point values and must be of the same data type. The output index values can be signed or unsigned integers.

`INDEX = step(H,...,CODEBOOK)` uses the codebook given in input `CODEBOOK`, a k-by-N matrix with N codewords each of length k. This option is available when the `CodebookSource` property is `Input port`.

`INDEX = step(H,...,WEIGHTS)` uses the input vector `WEIGHTS` to emphasize or de-emphasize certain input values when calculating the distortion measure. `WEIGHTS` must be a vector of length equal to the number of rows of `INPUT`. This option is available when the `DistortionMeasure` property is `Weighted squared error` and the `WeightsSource` property is `Input port`.

`[..., CODEWORD] = step(H, ...)` outputs the `CODEWORD` values that correspond to each index value when the `CodewordOutputPort` property is `true`.

`[..., QERR] = step(H, ...)` outputs the quantization error `QERR` for each input value when the `QuantizationErrorOutputPort` property is `true`.

Note The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

.

dsp.Window

Purpose   Window object

Description   The `Window` object applies a window to an input signal.

To apply a window to an input signal:

1 Define and set up your window. See "Construction" on page 5-1302.

2 Call `step` to apply the window according to the properties of `dsp.Window`. The behavior of `step` is specific to each object in the toolbox.

Construction   `H = dsp.Window` returns a window object, `H`, that applies a Hamming window with symmetric sampling.

`H = dsp.Window('`*PropertyName*`',`*PropertyValue*`, ...)` returns a window object, `H`, with each property set to the specified value.

`H = dsp.Window(WINDOW,'`*PropertyName*`',`*PropertyValue*`, ...)` returns a window object, `H`, with the `WindowFunction` property set to `WINDOW` and other properties set to the specified values.

Properties   **WindowFunction**

Type of window

Specify the type of window to apply as `Bartlett`, `Blackman`, `Boxcar`, `Chebyshev`, `Hamming` , `Hann`, `Hanning`, `Kaiser`, `Taylor`, `Triang`. This property is tunable. The default is `Hamming`.

WeightsOutputPort

Enable the output of window weights

Set this property to `true` to output the window weights. The weights are an M-by-1 vector with M equal to the first dimension of the input. The default is `false`.

StopbandAttenuation

Level of stopband attenuation in decibels

Specify the level of stopband attenuation in decibels. This property only applies when the `WindowFunction` property is `Chebyshev`. The default is 50. This property is tunable.

Beta

Kaiser window parameter

Specify the Kaiser window parameter as a real number. Increasing the absolute value of `Beta` widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. This property only applies when `WindowFunction` property is `Kaiser`. The default is 10. This property is tunable.

NumConstantSidelobes

Number of constant sidelobes

Specify the number of constant sidelobes as an integer greater than zero. This property only applies when `WindowFunction` property is `Taylor`. The default is 4. This property is tunable.

MaximumSidelobeLevel

Maximum sidelobe level relative to mainlobe

Specify, in decibels, the maximum sidelobe level relative to the mainlobe as a real number less than or equal to zero. The default is −30, which produces sidelobes with peaks 30 dB down from the mainlobe peak. This property only applies when `WindowFunction` property is `Taylor`. This property is tunable.

Sampling

Window sampling for generalized-cosine windows

Specify the window sampling for generalized-cosine windows as `Symmetric` or `Periodic`. This property only applies when `WindowFunction` property is `Blackman`, `Hamming`, `Hann`, or `Hanning`. This property is tunable.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set
`FullPrecisionOverride` to `true`, which is the default, the object
computes all internal arithmetic and output data types using full
precision rules. These rules provide the most accurate fixed-point
numerics. It also turns off the display of other fixed-point
properties because they do not apply individually. These rules
guarantee that no quantization occurs within the object. Bits are
added, as needed, to ensure that no roundoff or overflow occurs. If
you set `FullPrecisionOverride` to `false`, fixed-point data types
are controlled through individual fixed-point property settings.
For more information, see "Full Precision for Fixed-Point System
Objects".

RoundingMethod

Rounding method for fixed-point operations

Specify the rounding method as one of `Ceiling`, `Convergent`,
`Floor`, `Nearest`, `Round`, `Simplest`, or `Zero`. The default is `Floor`.
This property applies only if the object is not in full precision
mode.

OverflowAction

Overflow action for fixed-point operations

Specify the overflow action as one of `Wrap` or `Saturate`. The
default is `Wrap`. This property applies only if the object is not in
full precision mode.

WindowDataType

Window word and fraction lengths

Specify the window fixed-point data type as one of `Same word length as input` or `Custom`. The default is `Same word length as input`.

CustomWindowDataType

Window word and fraction lengths

Specify the window fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `WindowDataType` property to `Custom`. The default is `numerictype([],16,15)`.

ProductDataType

Product word and fraction lengths

Specify the product fixed-point data type as one of `Full precision`, `Same as input`, or `Custom`. The default is `Full precision`.

CustomProductDataType

Product word and fraction lengths

Specify the product fixed-point type as a scaled `numerictype` object with a `Signedness` of `Auto`. This property applies when you set the `ProductDataType` property to `Custom`. The default is `numerictype([],16,15)`.

OutputDataType

Output data type

Specify the output fixed-point data type as one of `Same as product`, `Same as input`, `Custom`. The default is `Same as product`.

CustomOutputDataType

Output word and fraction lengths

Specify the output fixed-point type as a `numerictype` object with a `Signedness` of `Auto`. This property applies when you

set the `OutputDataType` property to `Custom`. The default is
`numerictype([],16,15)`.

Methods

clone	Create window object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs of step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Multiply input by window

Examples

Apply Hamming window to input signal:

```
 hwin = dsp.Window( ...
'WindowFunction', 'Hamming', ...
'WeightsOutputPort',true);
 x = rand(64,1);
[y, w] = step(hwin, x);
% View the window's time and frequency domain responses
wvtool(w);
```

Algorithms

This object implements the algorithm, inputs, and outputs described
on the Window Function block reference page. The object properties
correspond to the block parameters, except:

- **Operation** — The window object does not support the `Generate window` option.

- **Operation** — The `Generate and apply window` option on the block
  corresponds to the `WeightsOutputPort` property set to `true` on the
  window object.

- The window object only supports frame-based processing.

See Also   dsp.FFT | sigwin.bartlett | sigwin.blackman | sigwin.chebwin | sigwin.hamming | sigwin.hann | sigwin.kaiser | sigwin.taylorwin | sigwin.triang | wvtool

dsp.Window.clone

Purpose   Create window object with same property values

Syntax   C = clone(H)

Description   C = clone(H) creates a window object, C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   N = getNumInputs(H) returns the number of expected inputs, N, to
                   the step method.

                   The getNumInputs method returns a positive integer that is the number
                   of expected inputs (not counting the object itself) to the step method.
                   This value will change if you alter any properties that turn inputs on or
                   off. You must call the step method with the number of input arguments
                   equal to the result of getNumInputs(*H*).

dsp.Window.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of the step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose　　　Locked status for input attributes and nontunable properties

Syntax　　　　isLocked(H)

Description　　isLocked(H) returns the locked state of the window object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

dsp.Window.release

Purpose   Allow property value and input characteristics changes

Syntax   release(H)

Description release(H) releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

Note You can use the release method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Multiply input by window

Syntax   Y = step(H,X)
                [Y, W] = step(H,X)

Description Y = step(H,X) generates the windowed output, Y , of the input, X, using the specified window.

[Y, W] = step(H,X) returns the window values W when the WeightsOutputPort property is true.

dsp.ZeroCrossingDetector

Purpose　Zero crossing detector

Description　The `ZeroCrossingDetector` object counts the number of times the signal crosses zero, or changes sign. The zero crossing detector supports both floating-point and fixed-point data types.

To count the number of times a signal crosses zero or changes sign:

1 Define and set up your zero crossing detector. See "Construction" on page 5-1314.

2 Call `step` to count the number of times according to the properties of `dsp.ZeroCrossingDetector`. The behavior of `step` is specific to each object in the toolbox.

Construction　`H = dsp.ZeroCrossingDetector` returns a zero crossing detector object, H, that counts the number of zero crossings in the real-valued,, floating-point, or fixed-point frame-based vector or matrix.

`H = dsp.ZeroCrossingDetector('`*PropertyName*`',`*PropertyValue*`, ...)` returns a zero crossing detector object, H, with each property set to the specified value.

Properties　**FrameBasedProcessing**

Enable frame-based processing

Set this property to `true` to enable frame-based processing. Set this property to `false` to enable sample-based processing. The default is `true`.

If the `FrameBasedProcessing` property is `true`:

- The zero crossing detector treats a column vector or the columns of a matrix as single, independent channels.

- The zero crossing detector treats a length *N* row vector as *N* independent channels.

If the `FrameBasedProcessing` property is `false`:

- The zero crossing detector treats each entry of a vector or matrix as an independent channel.

Methods		
	clone	Create zero crossing detection object with same property values
	getNumInputs	Number of expected inputs to step method
	getNumOutputs	Number of outputs of step method
	isLocked	Locked status for input attributes and nontunable properties
	release	Allow property value and input characteristics changes
	step	Count zero crossings in input

Examples   Find number of zero crossings in electrocardiogram data:

```
EcgData = ecg(500)';
Hzerocross = dsp.ZeroCrossingDetector;
NumZeroCross = step(Hzerocross,EcgData);  % Equal to 4
plot(1:500,EcgData,'b',[0 500],[0 0],'r','linewidth',2);
```

Algorithms   This object implements the algorithm, inputs, and outputs described on the Zero Crossing block reference page. The object properties correspond to the block parameters.

Both this object and its corresponding block let you specify whether to process inputs as individual samples or as frames of data. The object uses the FrameBasedProcessing property. The block uses the **Input processing** parameter. See "Set the FrameBasedProcessing Property of a System object" for more information.

dsp.ZeroCrossingDetector.clone

Purpose   Create zero crossing detection object with same property values

Syntax   C = clone(H)

Description C = clone(H) creates a zero crossing detector object, C, with the same property values as H. The clone method creates a new unlocked object.

Purpose   Number of expected inputs to step method

Syntax   N = getNumInputs(H)

Description   `N = getNumInputs(H)` returns the number of expected inputs, `N`, to
                 the `step` method.

                 The `getNumInputs` method returns a positive integer that is the number
                 of expected inputs (not counting the object itself) to the `step` method.
                 This value will change if you alter any properties that turn inputs on or
                 off. You must call the `step` method with the number of input arguments
                 equal to the result of `getNumInputs(H)`.

dsp.ZeroCrossingDetector.getNumOutputs

Purpose   Number of outputs of step method

Syntax   N = getNumOutputs(H)

Description   N = getNumOutputs(H) returns the number of outputs, N, of step method.

The getNumOutputs method returns a positive integer that is the number of outputs from the step method. This value will change if you alter any properties that turn outputs on or off.

Purpose   Locked status for input attributes and nontunable properties

Syntax   isLocked(H)

Description isLocked(H) returns the locked state of the zero crossing detector object.

The isLocked method returns a logical value that indicates whether input attributes and nontunable properties for the object are locked. The object performs an internal initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. After locking, the isLocked method returns a true value.

Purpose	Allow property value and input characteristics changes
Syntax	`release(H)`
Description	`release(H)` releases system resources, such as memory, file handles, and hardware connections, and lets you change any properties or input characteristics.

> **Note** You can use the `release` method on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

Purpose   Count zero crossings in input

Syntax   `Y = step(H,X)`

Description   `Y = step(H,X)` counts the number of zero crossings, `Y`, in the vector or matrix input `X`.

Note The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

matlab.System

Purpose   Base class for System objects

Description   `matlab.System` is the base class for System objects. In your class definition file, you must subclass your object from this base class (or from another class that derives from this base class). Subclassing allows you to use the implementation and service methods provided by this base class to build your object. You use this syntax as the first line of your class definition file to directly inherit from the `matlab.System` base class, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.System
```

Note You must set `Access=protected` for each `matlab.System` method you use in your code.

Methods

getDiscreteStateImpl	Discrete state property values
getNumInputsImpl	Number of input arguments passed to step and setup methods
getNumOutputsImpl	Number of outputs returned by method
isInactivePropertyImpl	Active or inactive flag for properties
loadObjectImpl	Load saved System object from MAT file
processTunedPropertiesImpl	Action when tunable properties change
releaseImpl	Release resources
resetImpl	Reset System object states
saveObjectImpl	Save System object in MAT file

setProperties	Set property values from name-value pair inputs
setupImpl	Initialize System object
stepImpl	System output and state update equations
validateInputsImpl	Validate inputs to step method
validatePropertiesImpl	Validate property values

Attributes   In addition to the attributes available for MATLAB objects, you can apply the following attributes to any property of a custom System object.

Nontunable	After an object is locked (after step or setup has been called), use Nontunable to prevent a user from changing that property value. By default, all properties are tunable. The Nontunable attribute is useful to lock a property that has side effects when changed. This attribute is also useful for locking a property value assumed to be constant during processing. You should always specifiy properties that affect the number of input or output ports as Nontunable.
Logical	Use Logical to limit the property value to a logical, scalar value. Any scalar value that can be converted to a logical is also valid, such as 0 or 1.
PositiveInteger	Use PositiveInteger to limit the property value to a positive integer value.
DiscreteState	Use DiscreteState to mark a property so it will display its state value when you use the getDiscreteState method.

To learn more about attributes, see "Property Attributes" in the MATLAB Object-Oriented Programming documentation.

Examples

Create a simple System object, `AddOne`, which subclasses from `matlab.System`. You place this code into a MATLAB file, `AddOne.m`.

```
classdef AddOne < matlab.System
%ADDONE Compute an output value that increments the input by one

    methods (Access=protected)
       % stepImpl method is called by the step method.
       function y = stepImpl(~,x)
           y = x + 1;
       end
    end
end
```

To use this object, create an instance of `AddOne`, provide an input, and use the `step` method:

```
hAdder = AddOne;
x = 1;
y = step(hAdder,x)
```

Assign the `Nontunable` attribute to the `InitialValue` property, which you define in your class definition file.

```
properties (Nontunable)
   InitialValue
end
```

See Also

matlab.system.StringSet | | | |
matlab.system.mixin.FiniteSource | | |

How To

· "Object-Oriented Programming"

- Class Attributes
- Property Attributes
- "Method Attributes"
- "Define Basic System Objects"
-
-
-
- "Define Property Attributes"
-
-
-

matlab.System.getDiscreteStateImpl

Purpose   Discrete state property values

Syntax   s = getDiscreteStateImpl(obj)

Description   s = getDiscreteStateImpl(obj) returns a struct s of state values.
The field names of the struct are the object's DiscreteState property
names. To restrict or change the values returned by getDiscreteState
method, you can override this getDiscreteStateImpl method. End
users cannot specify scaled double fi objects as inputs to discrete state
properties.

getDiscreteStatesImpl is called by the getDiscreteState method,
which is called by the setup method.

Note You must set Access=protected for this method.

**Input
Arguments**

obj

    System object handle

**Output
Arguments**

s

    Struct of state values.

Examples   methods (Access=protected)
                    function s = getDiscreteState(obj)
                    end
                 end

See Also   setupImpl | | |

How To   • "Define Property Attributes"

               •

               •

.

matlab.System.getNumInputsImpl

Purpose	Number of input arguments passed to step and setup methods
Syntax	num = getNumInputsImpl(obj)

Description   num = getNumInputsImpl(obj) returns the number of inputs num (excluding the System object handle) expected by the step method. The default implementation returns 1, which requires one input from the user, in addition to the System object handle. To specify a value other than 1, you must use include the getNumInputsImpl method in your class definition file.

getNumInputsImpl is called by the getNumInputs method and by the setup method if the number of inputs has not been determined already.

Note You must set Access=protected for this method.

Input Arguments

obj

    System object handle

Output Arguments

num

    Number of inputs expected by the step method for the specified object.

    Default: 1

Examples   Specify the number of inputs (2, in this case) expected by the step method.

```
methods (Access=protected)
   function num = getNumInputsImpl(obj)
      num = 2;
   end
end
```

Specify that the step method will not accept any inputs.

```
methods (Access=protected)
   function num = getNumInputsImpl(~)
      num = 0;
   end
end
```

See Also   setupImpl | | | | stepImpl | | | | getNumOutputsImpl | | |

How To   • "Change Number of Step Inputs or Outputs"

                 •

                 •

                 •

matlab.System.getNumOutputsImpl

Purpose	Number of outputs returned by step method
Syntax	num = getNumOutputsImpl (obj)
Description	num = getNumOutputsImpl (obj) returns the number of outputs from the step method. The default implementation returns 1 output. To specify a value other than 1, you must use include the getNumOutputsImpl method in your class definition file.
	getNumOutputsImpl is called by the getNumOutputs method, if the number of outputs has not been determined already.

> **Note** You must set Access=protected for this method.

Input Arguments	**obj**
	System object handle
Output Arguments	**num**
	Number of outputs to be returned by the step method for the specified object.
Examples	Specify the number of outputs (2, in this case) returned from the step method.

```
methods (Access=protected)
   function num = getNumOutputsImpl(obj)
      num = 2;
   end
end
```

Specify that the step method does not return any outputs.

```
methods (Access=protected)
```

```
      function num = getNumOutputsImpl(~)
         num = O;
      end
end
```

See Also   stepImpl | | | | getNumInputsImpl | | | | setupImpl | | |

How To   • "Change Number of Step Inputs or Outputs"

          •

          •

          •

matlab.System.isInactivePropertyImpl

Purpose	Active or inactive flag for properties
Syntax	`flag = isInactivePropertyImpl(obj,prop)`

Description `flag = isInactivePropertyImpl(obj,prop)` specifies whether a property is inactive for the current object configuration. An *inactive property* is a property that is not relevant to the object, given the values of other properties. Inactive properties are not shown if you use the `disp` method to display object properties. If you attempt to use public access to directly access or use `get` or `set` on an inactive property, a warning occurs.

`isInactiveProperty` is called by the `disp` method and by the `get` and `set` methods.

Note You must set `Access=protected` for this method.

Input Arguments

obj

System object handle

prop

Property name

Output Arguments

flag

Logical scalar value indicating whether the input property `prop` is inactive for the current object configuration.

Examples Display the `InitialValue` property only when the `UseRandomInitialValue` property value is `false`.

```
methods (Access=protected)
  function flag = isInactivePropertyImpl(obj,propertyName)
    if strcmp(propertyName,'InitialValue')
      flag = obj.UseRandomInitialValue;
```

```
        else
          flag = false;
        end
    end
end
```

See Also   setProperties | | |

How To   • "Hide Inactive Properties"

             •

             •

             •

matlab.System.loadObjectImpl

Purpose	Load saved System object from MAT file
Syntax	`loadObjectImpl(obj)`
Description	`loadObjectImpl(obj)` loads a saved System object, `obj`, from a MAT file. Your `loadObjectImpl` method should correspond to your `saveObjectImpl` method to ensure that all saved properties and data are loaded.
Input Arguments	**obj** System object handle

Examples   Load a saved System object. In this case, the object contains a child object, protected and private properties, and a discrete state.

```
methods(Access=protected)
  function loadObjectImpl(obj, s, wasLocked)
    % Load child System objects
    obj.child = matlab.System.loadObject(s.child);

    % Save protected & private properties
    obj.protected = s.protected;
    obj.pdependentprop = s.pdependentprop;

    % Save state only if locked when saved
    if wasLocked
      obj.state = s.state;
    end

    % Call base class method
    loadObjectImpl@matlab.System(obj,s,wasLocked);
  end
end
```

See Also   `saveObjectImpl`

How To   • "Load System Object"

• "Save System Object"

matlab.System.processTunedPropertiesImpl

Purpose   Action when tunable properties change

Syntax   processTunedPropertiesImpl(obj)

Description   processTunedPropertiesImpl(obj) specifies the actions to perform
                   when one or more tunable property values change. This method is
                   called as part of the next call to the step method after a tunable
                   property value changes. A property is tunable only if its Nontunable
                   attribute is false, which is the default.

                   processTunedPropertiesImpl is called by the step method.

                   Note You must set Access=protected for this method.

Tips   Use this method when a tunable property affects a different property
                   value. For example, two property values determine when to calculate
                   a lookup table. You want to perform that calculation when either
                   property changes. You also want the calculation to be done only once if
                   both properties change before the next call to the step method.

Input   **obj**
Arguments   System object handle

Examples   Use processTunedPropertiesImpl to recalculate the lookup table if
                   the value of either the NumNotes or MiddleC property changes.

```
methods (Access=protected)
  function processTunedPropertiesImpl(obj)
    % Generate a lookup table of note frequencies
    obj.pLookupTable = obj.MiddleC * (1+log(1:obj.NumNotes)/log(12));
  end
end
```

See Also   validatePropertiesImpl | | | | | setProperties | | |

How To
- "Validate Property and Input Values"
-
-
-
- "Define Property Attributes"
-
-
-

matlab.System.releaseImpl

Purpose   Release resources

Syntax   `releaseImpl(obj)`

Description   `releaseImpl(obj)` releases any resources used by the System object, such as file handles. This method also performs any necessary cleanup tasks. To release resources for a System object, you must use `releaseImpl` instead of a destructor.

`releaseImpl` is called by the `release` method. `releaseImpl` is also called when the object is deleted or cleared from memory, or when all references to the object have gone out of scope.

Note You must set `Access=protected` for this method.

Input Arguments

obj

System object handle

Examples   Use the `releaseImpl` method to close a file.

```
methods (Access=protected)
   function releaseImpl(obj)
     fclose(obj.pFileID);
   end
end
```

See Also   `resetImpl` | | |

How To   • "Release System Object Resources"

•

•

•

Purpose	Reset System object states
Syntax	`resetImpl(obj)`
Description	`resetImpl(obj)` defines the state reset equations for the System object. Typically you reset the states to a set of initial values.

`resetImpl` is called by the `reset` method. It is also called by the `setup` method, after the `setupImpl` method.

Note You must set `Access=protected` for this method.

Input Arguments	**obj**
	System object handle
Examples	Use the `reset` method to reset the counter `pCount` property to zero.

```
methods (Access=protected)
   function resetImpl(obj)
      obj.pCount = 0;
   end
end
```

See Also	`releaseImpl` | | |
How To	• "Reset Algorithm State"
	•
	•
	•

matlab.System.saveObjectImpl

Purpose   Save System object in MAT file

Syntax   saveObjectImpl(obj)

Description saveObjectImpl(obj) defines what System object obj property and
state values are saved in a MAT file when a user calls save on that
object. save calls saveObject, which then calls saveObjectImpl. If
you do not define a saveObjectImpl method for your System object
class, only public properties are saved. To save any private or protected
properties or state information, you must define a saveObjectImpl in
your class definition file.

You should save the state of an object only if the object is locked. When
the user loads that saved object, it loads in that locked state.

To save child object information, you use the associated saveObject
method within the saveObjectImpl method.

End users can use load, which calls loadObjectImpl to load a System
object into their workspace.

Input Arguments   **obj**

System object handle

Examples   Define what is saved for the System object. Call the base class version
of saveObjectImpl to save public properties. Then, save any child
System objects and any protected and private propertes. Finally, save
the state, if the object is locked.

```
methods(Access=protected)
    function s = saveObjectImpl(obj)
        s = saveObjectImpl@matlab.System(obj);
        s.child = matlab.System.saveObject(obj.child);
        s.protected = obj.protected;
        s.pdependentprop = obj.pdependentprop;
        if isLocked(obj)
            s.state = obj.state;
```

```
        end
    end
end
```

See Also   loadObjectImpl

How To   • "Save System Object"

               • "Load System Object"

matlab.System.setProperties

Purpose   Set property values from name-value pair inputs

Syntax   setProperties(obj,numargs,name1,value1,name2,value2,...)
                   setProperties(obj,numargs,arg1,...,argm,name1,value1,name2,
                       value2,...)

Description   setProperties(obj,numargs,name1,value1,name2,value2,...)
                   provides the name-value pair inputs to the System object constructor.
                   Use this syntax if every input must specify both name and value.

Note To allow standard name-value pair handling at construction,
define setProperties for your System object.

---

setProperties(obj,numargs,arg1,...,argm,name1,value1,name2,value2,...)
provides the value-only inputs, followed by the name-value pair inputs
to the System object during object construction. Use this syntax if you
want to allow users to specify one or more inputs by their values only.

**Input
Arguments**   **obj**

                   System object handle

                   numargs

                   Number of inputs passed in by the object constructor

                   name*

                   Name of property

                   value*

                   Value of the property

                   arg*

                   Value of property (for value-only input to the object constructor)

Examples   Set up the object so users can specify property values via name-value pairs when constructing the object.

```
methods
   function obj = MyFile(varargin)
      setProperties(obj,nargin,varargin{:});
   end
end
```

How To   • "Set Property Values at Construction Time"

•

•

•

matlab.System.setupImpl

Purpose   Initialize System object

Syntax   setupImpl(obj,input1, input2,...)

Description setupImpl(obj,input1, input2,...) sets up a System object. To acquire resources for a System object, you must use setupImpl instead of a constructor. setupImpl executes the first time the step method is called on an object after that object has been created. It also executes the next time step is called after an object has been released. . The number of inputs must match the number of inputs defined in the getNumInputsImpl method. You pass the inputs into setupImpl to use the input sizes, datatypes, etc. in the one-time calculations.

setupImpl is called by the setup method, which is done automatically as the first subtask of the step method on an unlocked System object.

Note You must set Access=protected for this method.

Tips   To validate properties or inputs use the validatePropertiesImpl, validateInputsImpl, or setProperties methods. Do not include validation in setupImpl.

Input Arguments

obj

  System object handle

input*

  Inputs to the setup method

Examples   Open a file for writing using the setupImpl method.

```
methods (Access=protected)
  function setupImpl(obj,data)
     obj.pFileID = fopen(obj.Filename, 'wb');
     if obj.pFileID < O
```

```
            error('Opening the file failed');
        end
    end
end
```

See Also   validatePropertiesImpl | | | | validateInputsImpl | | | |
                   setProperties | | |

How To   • "Initialize Properties and Setup One-Time Calculations"

                   •

                   •

                   •

                   • "Set Property Values at Construction Time"

                   •

                   •

                   •

matlab.System.stepImpl

Purpose	System output and state update equations
Syntax	[output1,output2,...] = stepImpl(obj,input1,input2,...)
Description	[output1,output2,...] = stepImpl(obj,input1,input2,...) defines the algorithm to execute when you call the step method on the specified object obj. The step method calculates the outputs and updates the object's state values using the inputs, properties, and state update equations.

stepImpl is called by the step method.

Note You must set Access=protected for this method.

Tips	The number of input arguments and output arguments must match the values returned by the getNumInputsImpl and getNumOutputsImpl methods, respectively
Input Arguments	**obj**
	System object handle
	input*
	Inputs to the step method
Output Arguments	**output**
	Output returned from the step method.
Examples	Use the stepImpl method to increment two numbers.

```
methods (Access=protected)
   function [y1,y2] = stepImpl(obj,x1,x2)
      y1 = x1 + 1;
      y2 = x2 + 1;
    end
```

See Also   getNumInputsImpl | | | | getNumOutputsImpl | | | |
validateInputsImpl | | |

How To
- "Define Basic System Objects"
-
-
-
- "Change Number of Step Inputs or Outputs"
-
-
-

matlab.System.validateInputsImpl

Purpose	Validate inputs to step method
Syntax	`validateInputsImpl(obj,input1,input2,...)`
Description	`validateInputsImpl(obj,input1,input2,...)` validates inputs to the `step` method at the beginning of initialization Validation includes checking data types, complexity, cross-input validation, and validity of inputs controlled by a property value.
	`validateInputsImpl` is called by the `setup` method before `setupImpl`. `validateInputsImpl` executes only once.

Note You must set `Access=protected` for this method.

Input Arguments

obj

    System object handle

input*

    Inputs to the `setup` method

Examples

Validate that the input is numeric.

```
methods (Access=protected)
   function validateInputsImpl(~,x)
      if ~isnumeric(x)
         error('Input must be numeric');
      end
   end
end
```

See Also `validatePropertiesImpl` | | | | | `setupImpl` | | |

How To • "Validate Property and Input Values"

    •

·

·

matlab.System.validatePropertiesImpl

Purpose   Validate property values

Syntax   validatePropertiesImpl(obj)

Description   validatePropertiesImpl(obj) validates interdependent or interrelated property values at the beginning of object initialization, such as checking that the dependent or related inputs are the same size.

validatePropertiesImpl is the first method called by the setup method. validatePropertiesImpl also is called before the processTunablePropertiesImpl method.

Note You must set Access=protected for this method.

Input Arguments

obj

   System object handle

Examples   Validate that the useIncrement property is true and that the value of the increment property is greater than zero.

```
methods (Access=protected)
   function validatePropertiesImpl(obj)
      if obj.useIncrement && obj.increment < 0
         error('The increment value must be positive');
      end
    end
end
```

See Also   processTunedPropertiesImpl | | | | setupImpl | | | | validateInputsImpl | | |

How To   • "Validate Property and Input Values"

   •

·
·

matlab.system.mixin.FiniteSource

Purpose   Finite source mixin class

Description `matlab.system.mixin.FiniteSource` is a class that defines the `isDone` method, which reports the state of a finite data source, such as an audio file.

To use this method, you must subclass from this class in addition to the `matlab.System` base class. You use the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```
classdef ObjectName < matlab.System &...
    matlab.system.mixin.FiniteSource
```

Methods   isDoneImpl   End-of-data flag

See Also   matlab.System | | |

Tutorials   • "Define Finite Source Objects"

                 •

                 •

                 •

How To   • "Object-Oriented Programming"

                 • Class Attributes

                 • Property Attributes

matlab.system.mixin.FiniteSource.isDoneImpl

Purpose   End-of-data flag

Syntax   status = isDoneImpl(obj)

Description status = isDoneImpl(obj) indicates if an end-of-data condition has
occurred. The isDone method should return false when data from a
finite source has been exhausted, typically by having read and output
all data from the source. You should also define the result of future
reads from an exhausted source in the isDoneImpl method.

isDoneImpl is called by the isDone method.

**Input
Arguments**   **obj**

    System object handle

**Output
Arguments**   **status**

    Logical value, true or false, that indicates if an end-of-data
    condition has occurred or not, respectively.

Examples   Set up isDoneImpl so the isDone method checks whether the object has
completed eight iterations.

```
methods (Access=private)
   function bdone = isDoneImpl(obj)
      bdone = obj.NumIters==8;
   end
end
```

See Also   matlab.system.mixin.FiniteSource | | |

How To   • "Define Finite Source Objects"

    •

    •

    •

matlab.system.StringSet

Purpose   Set of valid string values

Description   `matlab.system.StringSet` defines a list of valid string values for a property. This class validates the string in the property and enables tab completion for the property value. A *StringSet* allows only predefined or customized strings as values for the property.

A `StringSet` uses two linked properties, which you must define in the same class. One is a public property that contains the current string value. This public property is displayed to the user. The other property is a hidden property that contains the list of all possible string values. This hidden property should also have the transient attribute so its value is not saved to disk when you save the System object.

The following considerations apply when using StringSets:

- The string property that holds the current string can have any name.

- The property that holds the `StringSet` must use the same name as the string property with the suffix "Set" appended to it. The string set property is an instance of the `matlab.system.StringSet` class.

- Valid strings, defined in the `StringSet`, must be declared using a cell array. The cell array cannot be empty nor can it have any empty strings. Valid strings must be unique and are case-insensitive.

- The string property must be set to a valid `StringSet` value.

Examples   Set the string property, `Flavor`, and the `StringSet` property, `FlavorSet`, in this example.

```
properties
  Flavor='Chocolate';
end

properties (Hidden,Transient)
  FlavorSet = ...
    matlab.system.StringSet({'Vanilla','Chocolate'});
end
```

See Also   `matlab.System` | | |

How To
- "Object-Oriented Programming"
- Class Attributes
- Property Attributes
- "Limit Property Values to Finite String Set"
-
-
-

matlab.system.StringSet

6

Function Reference

Adaptive Filters

Least Mean Squares

`adaptfilt.adjlms`	FIR adaptive filter that uses adjoint LMS algorithm
`adaptfilt.blms`	FIR adaptive filter that uses BLMS
`adaptfilt.blmsfft`	FIR adaptive filter that uses FFT-based BLMS
`adaptfilt.dlms`	FIR adaptive filter that uses delayed LMS
`adaptfilt.filtxlms`	FIR adaptive filter that uses filtered-x LMS
`adaptfilt.lms`	FIR adaptive filter that uses LMS
`adaptfilt.nlms`	Normalized least mean squares adaptive filter
`adaptfilt.sd`	Sign-data FIR adaptive filter
`adaptfilt.se`	FIR adaptive filter that uses sign-error algorithm
`adaptfilt.ss`	FIR adaptive filter that uses sign-sign algorithm

Recursive Least Squares

`adaptfilt.ftf`	Fast transversal LMS adaptive filter
`adaptfilt.hrls`	FIR adaptive filter that uses householder (RLS)
`adaptfilt.hswrls`	FIR adaptive filter that uses householder sliding window RLS
`adaptfilt.qrdrls`	FIR adaptive filter that uses QR-decomposition-based RLS
`adaptfilt.rls`	Recursive least-squares FIR adaptive filter
`adaptfilt.swftf`	FIR adaptive filter that uses sliding window fast transversal least squares
`adaptfilt.swrls`	FIR adaptive filter that uses window recursive least squares (RLS)

Affine Projection

`adaptfilt.ap`	FIR adaptive filter that uses direct matrix inversion
`adaptfilt.apru`	FIR adaptive filter that uses recursive matrix updating
`adaptfilt.bap`	FIR adaptive filter that uses block affine projection

Frequency Domain

`adaptfilt.fdaf`	FIR adaptive filter that uses frequency-domain with bin step size normalization
`adaptfilt.pbfdaf`	Partitioned block frequency-domain FIR adaptive filter

`adaptfilt.pbufdaf`	Partitioned block unconstrained frequency-domain adaptive filter
`adaptfilt.tdafdct`	Adaptive filter that uses discrete cosine transform
`adaptfilt.tdafdft`	Adaptive filter that uses discrete Fourier transform
`adaptfilt.ufdaf`	FIR adaptive filter that uses unconstrained frequency-domain with quantized step size normalization

Lattice

`adaptfilt.gal`	FIR adaptive filter that uses gradient lattice
`adaptfilt.lsl`	Least squares lattice adaptive filter
`adaptfilt.qrdlsl`	QR-decomposition-based least-squares lattice adaptive filter

Discrete-Time Filters

`dfilt`	Discrete-time filter
`dfilt.allpass`	Allpass filter
`dfilt.calattice`	Coupled-allpass, lattice filter
`dfilt.calatticepc`	Coupled-allpass, power-complementary lattice filter
`dfilt.cascade`	Cascade of discrete-time filters
`dfilt.cascadeallpass`	Cascade of allpass discrete-time filters
`dfilt.cascadewdfallpass`	Cascade allpass WDF filters to construct allpass WDF
`dfilt.delay`	Delay filter
`dfilt.df1`	Discrete-time, direct-form I filter
`dfilt.df1sos`	Discrete-time, SOS direct-form I filter
`dfilt.df1t`	Discrete-time, direct-form I transposed filter
`dfilt.df1tsos`	Discrete-time, SOS direct-form I transposed filter
`dfilt.df2`	Discrete-time, direct-form II filter
`dfilt.df2sos`	Discrete-time, SOS, direct-form II filter
`dfilt.df2t`	Discrete-time, direct-form II transposed filter
`dfilt.df2tsos`	Discrete-time, SOS direct-form II transposed filter
`dfilt.dfasymfir`	Discrete-time, direct-form antisymmetric FIR filter
`dfilt.dffir`	Discrete-time, direct-form FIR filter

`dfilt.dffirt`	Discrete-time, direct-form FIR transposed filter
`dfilt.dfsymfir`	Discrete-time, direct-form symmetric FIR filter
`dfilt.farrowfd`	Fractional Delay Farrow filter
`dfilt.farrowlinearfd`	Farrow Linear Fractional Delay filter
`dfilt.fftfir`	Discrete-time, overlap-add, FIR filter
`dfilt.latticeallpass`	Discrete-time, lattice allpass filter
`dfilt.latticear`	Discrete-time, lattice, autoregressive filter
`dfilt.latticearma`	Discrete-time, lattice, autoregressive, moving-average filter
`dfilt.latticemamax`	Discrete-time, lattice, moving-average filter with maximum phase
`dfilt.latticemamin`	Discrete-time, lattice, moving-average filter with minimum phase
`dfilt.parallel`	Discrete-time, parallel structure filter
`dfilt.scalar`	Discrete-time, scalar filter
`dfilt.wdfallpass`	Wave digital allpass filter

Filter Specifications — Response Types

`fdesign`	Filter specification object
`fdesign.arbgrpdelay`	Arbitrary group delay filter specification object
`fdesign.arbmag`	Arbitrary response magnitude filter specification object
`fdesign.arbmagnphase`	Arbitrary response magnitude and phase filter specification object
`fdesign.audioweighting`	Audio weighting filter specification object
`fdesign.bandpass`	Bandpass filter specification object
`fdesign.bandstop`	Bandstop filter specification object
`fdesign.ciccomp`	CIC compensator filter specification object
`fdesign.comb`	IIR comb filter specification object
`fdesign.decimator`	Decimator filter specification object
`fdesign.differentiator`	Differentiator filter specification object
`fdesign.fracdelay`	Fractional delay filter specification object
`fdesign.halfband`	Halfband filter specification object
`fdesign.highpass`	Highpass filter specification object
`fdesign.hilbert`	Hilbert filter specification object
`fdesign.interpolator`	Interpolator filter specification
`fdesign.isinchp`	Inverse sinc highpass filter specification
`fdesign.isinclp`	Inverse sinc lowpass filter specification
`fdesign.lowpass`	Lowpass filter specification
`fdesign.notch`	Notch filter specification

`fdesign.nyquist`	Nyquist filter specification
`fdesign.octave`	Octave filter specification
`fdesign.parameq`	Parametric equalizer filter specification
`fdesign.peak`	Peak filter specification
`fdesign.polysrc`	Construct polynomial sample-rate converter (POLYSRC) filter designer
`fdesign.pulseshaping`	Pulse-shaping filter specification object
`fdesign.rsrc`	Rational-factor sample-rate converter specification

Filter Specifications — Design Methods

butter	Butterworth IIR filter design using specification object
cheby1	Chebyshev Type I filter using specification object
cheby2	Chebyshev Type II filter using specification object
design	Apply design method to filter specification object
designmethods	Methods available for designing filter from specification object
designopts	Valid input arguments and values for specification object and method
ellip	Elliptic filter using specification object
equiripple	Equiripple single-rate or multirate FIR filter from specification object
fircls	FIR Constrained Least Squares filter
ifir	Interpolated FIR filter from filter specification
iirlinphase	Quasi-linear phase IIR filter from halfband filter specification
kaiserwin	Kaiser window filter from specification object
maxflat	Maxflat FIR filter
multistage	Multistage filter from specification object
validstructures	Structures for specification object with design method
window	FIR filter using windowed impulse response

Multirate Filters

`mfilt.cascade`	Cascade filter objects
`mfilt.cicdecim`	Fixed-point CIC decimator
`mfilt.cicinterp`	Fixed-point CIC interpolator
`mfilt.farrowsrc`	Sample rate converter with arbitrary conversion factor
`mfilt.fftfirinterp`	Overlap-add FIR polyphase interpolator
`mfilt.firdecim`	Direct-form FIR polyphase decimator
`mfilt.firfracdecim (Obsolete)`	Direct-form FIR polyphase fractional decimator
`mfilt.firfracinterp (Obsolete)`	Direct-form FIR polyphase fractional interpolator
`mfilt.firinterp`	FIR filter-based interpolator
`mfilt.firsrc`	Direct-form FIR polyphase sample rate converter
`mfilt.firtdecim`	Direct-form transposed FIR filter
`mfilt.holdinterp`	FIR hold interpolator
`mfilt.iirdecim`	IIR decimator
`mfilt.iirinterp`	IIR interpolator
`mfilt.iirwdfdecim`	IIR wave digital filter decimator
`mfilt.iirwdfinterp`	IIR wave digital filter interpolator
`mfilt.linearinterp`	Linear interpolator

GUIs

fdatool	Open Filter Design and Analysis Tool
filterbuilder	GUI-based filter design

Filter Analysis

autoscale	Automatic dynamic range scaling
block	Generate block from multirate or multistage filter
coeffs	Coefficients for filters
cost	Estimate cost of discrete-time or multirate filter
cumsec	Vector of SOS filters for cumulative sections
denormalize	Undo filter coefficient and gain changes caused by normalize
disp	Filter properties and values
double	Cast fixed-point filter to use double-precision arithmetic
euclidfactors	Euclid factors for multirate filter
fftcoeffs	Frequency-domain coefficients
filter	Filter data with filter object
filtstates.cic	Store CIC filter states
firtype	Type of linear phase FIR filter
freqrespest	Frequency response estimate via filtering
freqrespopts	Options for filter frequency response analysis
freqsamp	Real or complex frequency-sampled FIR filter from specification object
freqz	Frequency response of filter
grpdelay	Filter group delay
help	Help for design method with filter specification
impz	Filter impulse response

isfir	Determine whether filter is FIR
islinphase	Determine whether filter is linear phase
ismaxphase	Determine whether filter is maximum phase
isminphase	Determine whether filter is minimum phase
isreal	Determine whether filter uses real coefficients
isstable	Determine whether filter is stable
limitcycle	Response of single-rate, fixed-point IIR filter
maxstep	Maximum step size for adaptive filter convergence
measure	Measure filter magnitude response
msepred	Predicted mean-squared error for adaptive filter
msesim	Measured mean-squared error for adaptive filter
noisepsd	Power spectral density of filter output due to roundoff noise
noisepsdopts	Options for running filter output noise PSD
norm	P-norm of filter
normalize	Normalize filter numerator or feed-forward coefficients
normalizefreq	Switch filter specification between normalized frequency and absolute frequency
nstates	Number of filter states
order	Order of fixed-point filter

phasedelay	Phase delay of filter
phasez	Unwrapped phase response for filter
polyphase	Polyphase decomposition of multirate filter
qreport	Most recent fixed-point filtering operation report
realizemdl	Simulink subsystem block for filter
reffilter	Reference filter for fixed-point or single-precision filter
reorder	Rearrange sections in SOS filter
reset	Reset filter properties to initial conditions
scale	Scale sections of SOS filter
scalecheck	Check scaling of SOS filter
set2int	Configure filter for integer filtering
setspecs	Specifications for filter specification object
specifyall	Fixed-point scaling modes in direct-form FIR filter
stepz	Step response for filter
sysobj	Create filter System object from discrete-time filter or multirate filter
zerophase	Zero-phase response for filter
zplane	Zero-pole plot for filter

To see the full listing of analysis methods that apply to the adaptfilt, dfilt, or mfilt objects, enter help adaptfilt, help dfilt, or help mfilt at the MATLAB prompt.

Fixed-Point Filters

cell2sos	Convert cell array to SOS matrix
get	Properties of quantized filter
isreal	Test if filter coefficients are real
reset	Reset properties of quantized filter to initial values
scale	Scale sections of SOS filters
scalecheck	Check scaling of SOS filter
scaleopts	Scaling options for second-order section scaling
set	Properties of quantized filter
sos	Convert quantized filter to SOS form, order, and scale
sos2cell	Convert SOS matrix to cell array

Quantized Filter Analysis

freqz	Frequency response of filter
impz	Filter impulse response
isallpass	Determine whether filter is allpass
isfir	Determine whether filter is FIR
islinphase	Determine whether filter is linear phase
ismaxphase	Determine whether filter is maximum phase
isminphase	Determine whether filter is minimum phase
isreal	Determine whether filter uses real coefficients
issos	Determine whether filter is SOS form
isstable	Determine whether filter is stable
noisepsd	Power spectral density of filter output due to roundoff noise
noisepsdopts	Options for running filter output noise PSD
zplane	Zero-pole plot for filter

SOS Conversion

`cell2sos`	Convert a cell array to a second-order sections matrix
`sos`	Convert a quantized filter to second-order sections form, order, and scale
`sos2cell`	Convert a second-order sections matrix to a cell array

Filter Design

fdatool	Open Filter Design and Analysis Tool
filterbuilder	GUI-based filter design
fircband	Constrained-band equiripple FIR filter
firceqrip	Constrained equiripple FIR filter
fireqint	Equiripple FIR interpolators
firgr	Parks-McClellan FIR filter
firhalfband	Halfband FIR filter
firlpnorm	Least P-norm optimal FIR filter
firls	Least-square linear-phase FIR filter design
firminphase	Minimum-phase FIR spectral factor
firnyquist	Lowpass Nyquist (Lth-band) FIR filter
ifir	Interpolated FIR filter from filter specification
iircomb	IIR comb notch or peak filter
iirgrpdelay	Optimal IIR filter with prescribed group-delay
iirlpnorm	Least P-norm optimal IIR filter
iirlpnormc	Constrained least Pth-norm optimal IIR filter
iirnotch	Second-order IIR notch filter
iirpeak	Second-order IIR peak or resonator filter

Filter Conversion

`ca2tf`	Convert coupled allpass filter to transfer function form
`cl2tf`	Convert coupled allpass lattice to transfer function form
`convert`	Convert filter structure of discrete-time or multirate filter
`firlp2hp`	Convert FIR lowpass filter to Type I FIR highpass filter
`firlp2lp`	Convert FIR Type I lowpass to FIR Type 1 lowpass with inverse bandwidth
`iirlp2bp`	Transform IIR lowpass filter to IIR bandpass filter
`iirlp2bs`	Transform IIR lowpass filter to IIR bandstop filter
`iirlp2hp`	Transform lowpass IIR filter to highpass filter
`iirlp2lp`	Transform lowpass IIR filter to different lowpass filter
`iirpowcomp`	Power complementary IIR filter
`set2int`	Configure filter for integer filtering
`tf2ca`	Transfer function to coupled allpass
`tf2cl`	Transfer function to coupled allpass lattice

Signal Logging

`dsp.util.getLogsArray`	Return logged signal as MATLAB array
`dsp.util.getSignalPath`	Paths to logged signals
`dsp.util.SignalPath`	Properties of paths to signals

Simulink Signal Processing Functions

dsplib	Open top-level DSP System Toolbox library
dspstartup	Configure Simulink environment for signal processing systems
dsp_links	Identify whether blocks in model are current, deprecated, or obsolete
liblinks	Check model for blocks from specific DSP System Toolbox libraries
rebuffer_delay	Number of samples of delay introduced by buffering and unbuffering operations

Functions — Alphabetical List

adaptfilt

Purpose　　Adaptive filter

Syntax　　`ha = adaptfilt.`*`algorithm`*`('input1',input2,...)`

Description　　`ha = adaptfilt.`*`algorithm`*`('input1',input2,...)` returns the
adaptive filter object `ha` that uses the adaptive filtering technique
specified by *algorithm*. When you construct an adaptive filter object,
include an *algorithm* specifier to implement a specific adaptive filter.
Note that you do not enclose the algorithm option in single quotation
marks as you do for most strings. To construct an adaptive filter object
you must supply an *algorithm* string — there is no default algorithm,
although every constructor creates a default adaptive filter when you do
not provide input arguments such as `input1` or `input2` in the calling
syntax.

Algorithms

For adaptive filter (`adaptfilt`) objects, the *algorithm* string
determines which adaptive filter algorithm your `adaptfilt` object
implements. Each available algorithm entry appears in one of the tables
along with a brief description of the algorithm. Click on the algorithm in
the first column to get more information about the associated adaptive
filter technique.

- "Least Mean Squares (LMS) Based FIR Adaptive Filters" on page 7-3
- "Recursive Least Squares (RLS) Based FIR Adaptive Filters" on page 7-4
- "Affine Projection (AP) FIR Adaptive Filters" on page 7-4
- "FIR Adaptive Filters in the Frequency Domain (FD)" on page 7-5
- "Lattice Based (L) FIR Adaptive Filters" on page 7-5

Least Mean Squares (LMS) Based FIR Adaptive Filters

adaptfilt.algorithm String	Algorithm Used to Generate Filter Coefficients
adaptfilt.adjlms	Use the Adjoint LMS FIR adaptive filter algorithm
adaptfilt.blms	Use the Block LMS FIR adaptive filter algorithm
adaptfilt.blmsfft	Use the FFT-based Block LMS FIR adaptive filter algorithm
adaptfilt.dlms	Use the delayed LMS FIR adaptive filter algorithm
adaptfilt.filtxlms	Use the filtered-x LMS FIR adaptive filter algorithm
adaptfilt.lms	Use the LMS FIR adaptive filter algorithm
adaptfilt.nlms	Use the normalized LMS FIR adaptive filter algorithm
adaptfilt.sd	Use the sign-data LMS FIR adaptive filter algorithm
adaptfilt.se	Use the sign-error LMS FIR adaptive filter algorithm
adaptfilt.ss	Use the sign-sign LMS FIR adaptive filter algorithm

For further information about an adapting algorithm, refer to the reference page for the algorithm.

Recursive Least Squares (RLS) Based FIR Adaptive Filters

adaptfilt.algorithm String	Algorithm Used to Generate Filter Coefficients
adaptfilt.ftf	Use the fast transversal least squares adaptation algorithm
adaptfilt.qrdrls	Use the QR-decomposition RLS adaptation algorithm
adaptfilt.hrls	Use the householder RLS adaptation algorithm
adaptfilt.hswrls	Use the householder SWRLS adaptation algorithm
adaptfilt.rls	Use the recursive-least squares (RLS) adaptation algorithm
adaptfilt.swrls	Use the sliding window (SW) RLS adaptation algorithm
adaptfilt.swftf	Use the sliding window FTF adaptation algorithm

For more complete information about an adapting algorithm, refer to the reference page for the algorithm.

Affine Projection (AP) FIR Adaptive Filters

adaptfilt.algorithm String	Algorithm Used to Generate Filter Coefficients
adaptfilt.ap	Use the affine projection algorithm that uses direct matrix inversion
adaptfilt.apru	Use the affine projection algorithm that uses recursive matrix updating
adaptfilt.bap	Use the block affine projection adaptation algorithm

To find more information about an adapting algorithm, refer to the reference page for the algorithm.

FIR Adaptive Filters in the Frequency Domain (FD)

adaptfilt.algorithm String	Algorithm Used to Generate Filter Coefficients
adaptfilt.fdaf	Use the frequency domain adaptation algorithm
adaptfilt.pbfdaf	Use the partition block version of the FDAF algorithm
adaptfilt.pbufdaf	Use the partition block unconstrained version of the FDAF algorithm
adaptfilt.tdafdct	Use the transform domain adaptation algorithm using DCT
adaptfilt.tdafdft	Use the transform domain adaptation algorithm using DFT
adaptfilt.ufdaf	Use the unconstrained FDAF algorithm for adaptation

For more information about an adapting algorithm, refer to the reference page for the algorithm.

Lattice Based (L) FIR Adaptive Filters

adaptfilt.algorithm String	Algorithm Used to Generate Filter Coefficients
adaptfilt.gal	Use the gradient adaptive lattice filter adaptation algorithm
adaptfilt.lsl	Use the least squares lattice adaptation algorithm
adaptfilt.qrdlsl	Use the QR decomposition least squares lattice adaptation algorithm

For more information about an adapting algorithm, refer to the reference page for the algorithm.

Properties for All Adaptive Filter Objects

Each reference page for an algorithm and `adaptfilt.algorithm` object specifies which properties apply to the adapting algorithm and how to use them.

Methods for Adaptive Filter Objects

As is true with all objects, methods enable you to perform various operations on `adaptfilt` objects. To use the methods, you apply them to the object handle that you assigned when you constructed the `adaptfilt` object.

Most of the analysis methods that apply to `dfilt` objects also work with `adaptfilt` objects. Methods like `freqz` rely on the filter coefficients in the `adaptfilt` object. Since the coefficients change each time the filter adapts to data, you should view the results of using a method as an analysis of the filter at a moment in time for the object. Use caution when you apply an analysis method to your adaptive filter objects — always check that your result approached your expectation.

In particular, the Filter Visualization Tool (FVTool) supports all of the `adaptfilt` objects. Analyzing and viewing your `adaptfilt` objects is straightforward — use the `fvtool` method with the name of your object

```
fvtool(objectname)
```

to launch FVTool and work with your object.

Some methods share their names with functions in Signal Processing Toolbox software, or even functions in this toolbox. Functions that share names with methods behave in a similar way. Using the same name for more than one function or method is called *overloading* and is common in many toolboxes.

Method	Description
adaptfilt/coefficients	Return the instantaneous adaptive filter coefficients
adaptfilt/filter	Apply an adaptfilt object to your signal
adaptfilt/freqz	Plot the instantaneous adaptive filter frequency response
adaptfilt/grpdelay	Plot the instantaneous adaptive filter group delay
adaptfilt/impz	Plot the instantaneous adaptive filter impulse response.
adaptfilt/info	Return the adaptive filter information.
adaptfilt/isfir	Test whether an adaptive filter is an finite impulse response (FIR) filters.
adaptfilt/islinphase	Test whether an adaptive filter is linear phase
adaptfilt/ismaxphase	Test whether an adaptive filter is maximum phase
adaptfilt/isminphase	Test whether an adaptive filter is minimum phase
adaptfilt/isreal	True whether an adaptive filter has real coefficients
adaptfilt/isstable	Test whether an adaptive filter is stable
adaptfilt/maxstep	Return the maximum step size for an adaptive filter
adaptfilt/msepred	Return the predicted mean square error
adaptfilt/msesim	Return the measured mean square error via simulation.

Method	Description
adaptfilt/phasez	Plot the instantaneous adaptive filter phase response
adaptfilt/reset	Reset an adaptive filter to initial conditions
adaptfilt/stepz	Plot the instantaneous adaptive filter step response
adaptfilt/tf	Return the instantaneous adaptive filter transfer function
adaptfilt/zerophase	Plot the instantaneous adaptive filter zerophase response
adaptfilt/zpk	Return a matrix containing the instantaneous adaptive filter zero, pole, and gain values
adaptfilt/zplane	Plot the instantaneous adaptive filter in the Z-plane

Working with Adaptive Filter Objects

The next sections cover viewing and changing the properties of adaptfilt objects. Generally, modifying the properties is the same for adaptfilt, dfilt, and mfilt objects and most of the same methods apply to all.

Viewing Object Properties

As with any object, you can use get to view a adaptfilt object's properties. To see a specific property, use

```
get(ha,'property')
```

To see all properties for an object, use

```
get(ha)
```

Changing Object Properties

To set specific properties, use

```
set(ha,'property1',value1,'property2',value2,...)
```

You must use single quotation marks around the property name so MATLAB treats them as strings.

Copying an Object

To create a copy of an object, use `copy`.

```
ha2 = copy(ha)
```

Note Using the syntax `ha2 = ha` copies only the object handle and does not create a new object — `ha` and `ha2` are not independent. When you change the characteristics of `ha2`, those of `ha` change as well.

Using Filter States

Two properties control your adaptive filter states.

- `States` — stores the current states of the filter. Before the filter is applied, the states correspond to the initial conditions and after the filter is applied, the states correspond to the final conditions.

- `PersistentMemory` — resets the filter before filtering. The default value is `false` which causes the properties that are modified by the filter, such as `coefficients` and `states`, to be reset to the value you specified when you constructed the object, before you use the object to filter data. Setting `PersistentMemory` to `true` allows the object to retain its current properties between filtering operations, rather than resetting the filter to its property values at construction.

Examples   Construct an LMS adaptive filter object and use it to identify an unknown system. For this example, use 500 iteration of the adapting process to determine the unknown filter coefficients. Using the LMS

algorithm represents one of the most straightforward technique for adaptive filters.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 0.008;            % LMS step size.
ha = adaptfilt.lms(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');  grid on;
```


See Also   dfilt | filter | mfilt

Purpose   FIR adaptive filter that uses adjoint LMS algorithm

Syntax   ha = adaptfilt.adjlms(l,step,leakage,pathcoeffs,pathest,...
                 errstates,pstates,coeffs,states)

Description ha = adaptfilt.adjlms(l,step,leakage,pathcoeffs,pathest,...
                 errstates,pstates,coeffs,states) constructs object ha, an FIR
                 adjoint LMS adaptive filter. l is the adaptive filter length (the number
                 of coefficients or taps) and must be a positive integer. l defaults to
                 10 when you omit the argument. step is the adjoint LMS step size.
                 It must be a nonnegative scalar. When you omit the step argument,
                 step defaults to 0.1.

                 leakage is the adjoint LMS leakage factor. It must be a scalar between
                 0 and 1. When leakage is less than one, you implement a leaky version
                 of the adjlms algorithm to determine the filter coefficients. leakage
                 defaults to 1 specifying no leakage in the algorithm.

                 pathcoeffs is the secondary path filter model. This vector should
                 contain the coefficient values of the secondary path from the output
                 actuator to the error sensor.

                 pathest is the estimate of the secondary path filter model. pathest
                 defaults to the values in pathcoeffs.

                 errstates is a vector of error states of the adaptive filter. It must have
                 a length equal to the filter order of the secondary path model estimate.
                 errstates defaults to a vector of zeros of appropriate length. pstates
                 contains the secondary path FIR filter states. It must be a vector of
                 length equal to the filter order of the secondary path model. pstates
                 defaults to a vector of zeros of appropriate length. The initial filter
                 coefficients for the secondary path filter compose vector coeffs. It must
                 be a length l vector. coeffs defaults to a length l vector of zeros.
                 states is a vector containing the initial filter states. It must be a vector
                 of length l+ne-1, where ne is the length of errstates. When you omit
                 states, it defaults to an appropriate length vector of zeros.

adaptfilt.adjlms

Properties In the syntax for creating the `adaptfilt` object, the input options are properties of the object created. This table lists the properties for the adjoint LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
Algorithm	None	Specifies the adaptive filter algorithm the object uses during adaptation
Coefficients	Length l vector with zeros for all elements	Adjoint LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need l entries in `coefficients`. Updated filter coefficients are returned in `coefficients` when you use s as an output argument.
ErrorStates	[0,...,0]	A vector of the error states for your adaptive filter, with length equal to the order of your secondary path filter.
FilterLength	10	The number of coefficients in your adaptive filter.
Leakage	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
SecondaryPathCoeffs	No default	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor.
SecondaryPathEstimate	pathcoeffs values	An estimate of the secondary path filter model.

Property	Default Value	Description
SecondaryPathStates	Length of the secondary path filter. All elements are zeros.	The states of the secondary path filter, the unknown system
States	l+ne+1, where ne is length(errstates)	Contains the initial conditions for your adaptive filter and returns the states of the FIR filter after adaptation. If omitted, it defaults to a zero vector of length equal to l+ne+1. When you use adaptfilt.adjlms in a loop structure, use this element to specify the initial filter states for the adapting FIR filter.
Stepsize	0.1	Sets the adjoint LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

Examples   Demonstrate active noise control of a random noise signal that runs for 1000 samples.

```
x = randn(1,1000);      % Noise source
g = fir1(47,0.4);       % FIR primary path system model
```

adaptfilt.adjlms

```
n  = 0.1*randn(1,1000);  % Observation noise signal
d  = filter(g,1,x)+n;    % Signal to be canceled (desired)
b  = fir1(31,0.5);       % FIR secondary path system model
mu = 0.008;              % Adjoint LMS step size
ha = adaptfilt.adjlms(32,mu,1,b);
[y,e] = filter(ha,x,d);
plot(1:1000,d,'b',1:1000,e,'r');
title('Active Noise Control of a Random Noise Signal');
legend('Original','Attenuated');
xlabel('Time Index'); ylabel('Signal Value');  grid on;
```

Reviewing the figure shows that the adaptive filter attenuates the original noise signal as you expect.

References   Wan, Eric., "Adjoint LMS: An Alternative to Filtered-X LMS and Multiple Error LMS," Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP), pp. 1841-1845, 1997

See Also   `adaptfilt.dlms` | `adaptfilt.filtxlms`

Purpose   FIR adaptive filter that uses direct matrix inversion

Syntax   ha = adaptfilt.ap(l,step,projectord,offset,coeffs,states,...
                  errstates,epsstates)

Description   ha =
                  adaptfilt.ap(l,step,projectord,offset,coeffs,states,...
                  errstates,epsstates) constructs an affine projection FIR adaptive
                  filter ha using direct matrix inversion.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.ap.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. l defaults to 10.
step	Affine projection step size. This is a scalar that should be a value between zero and one. Setting step equal to one provides the fastest convergence during adaptation. step defaults to 1.
projectord	Projection order of the affine projection algorithm. projectord defines the size of the input signal covariance matrix and defaults to two.
offset	Offset for the input signal covariance matrix. You should initialize the covariance matrix to a diagonal matrix whose diagonal entries are equal to the offset you specify. offset should be positive. offset defaults to one.

Input Argument	Description
coeffs	Vector containing the initial filter coefficients. It must be a length l vector, the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
states	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).
errstates	Vector of the adaptive filter error states. errstates defaults to a zero vector with length equal to (projectord - 1).
epsstates	Vector of the epsilon values of the adaptive filter. epsstates defaults to a vector of zeros with (projectord - 1) elements.

Properties   Since your adaptfilt.ap filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.ap objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps

Name	Range	Description
ProjectionOrder	1 to as large as needed.	Projection order of the affine projection algorithm. ProjectionOrder defines the size of the input signal covariance matrix and defaults to two.
OffsetCov	Matrix of values	Contains the offset covariance matrix
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector, the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).
ErrorStates	Vector of elements	Vector of the adaptive filter error states. errstates defaults to a zero vector with length equal to (projectord - 1).
EpsilonStates	Vector of elements	Vector of the epsilon values of the adaptive filter. epsstates defaults to a vector of zeros with (projectord - 1) elements.

Name	Range	Description
StepSize	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to true.

Examples   Quadrature phase shift keying (QPSK) adaptive equalization using a 32-coefficient FIR filter. Run the adaptation for 1000 iterations.

```
D  = 16;                    % Number of samples of delay
b  = exp(j*pi/4)*[-0.7 1];  % Numerator coefficients of channel
a  = [1 -0.7];              % Denominator coefficients of channel
ntr= 1000;                  % Number of iterations
s  = sign(randn(1,ntr+D)) + j*sign(randn(1,ntr+D));% Baseband Signal
n  = 0.1*(randn(1,ntr+D) + j*randn(1,ntr+D));      % Noise signal
r  = filter(b,a,s)+n;       % Received signal
x  = r(1+D:ntr+D);    % Input signal (received signal)
d  = s(1:ntr);        % Desired signal (delayed QPSK signal)
mu = 0.1;             % Step size
po = 4;               % Projection order
offset = 0.05;        % Offset for covariance matrix
ha = adaptfilt.ap(32,mu,po,offset);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e])); title('In-Phase Components');
```

```
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');
axis([-3 3 -3 3]); title('Received Signal Scatter Plot');
axis('square');  xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

The four plots shown reveal the QPSK process at work.

References   [1] Ozeki, K. and Umeda, T., "An Adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and Its Properties," Electronics and Communications in Japan, vol.67-A, no. 5, pp. 19-27, May 1984

[2] Maruyama, Y., "A Fast Method of Projection Algorithm," Proc. 1990 IEICE Spring Conf., B-744

See Also   msesim

Purpose   FIR adaptive filter that uses recursive matrix updating

Syntax   ha = adaptfilt.apru(l,step,projectord,offset,coeffs,states,
               ...errstates,epsstates)

Description   ha = adaptfilt.apru(l,step,projectord,offset,coeffs,states,
                 ...errstates,epsstates) constructs an affine projection FIR adaptive
                 filter ha using recursive matrix updating.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.apru.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps). It must be a positive integer. l defaults to 10.
step	Affine projection step size. This is a scalar that should be a value between zero and one. Setting step equal to one provides the fastest convergence during adaptation. step defaults to 1.
projectord	Projection order of the affine projection algorithm. projectord defines the size of the input signal covariance matrix and defaults to two.
offset	Offset for the input signal covariance matrix. You should initialize the covariance matrix to a diagonal matrix whose diagonal entries are equal to the offset you specify. offset should be positive. offset defaults to one.
coeffs	Vector containing the initial filter coefficients. It must be a length l vector, the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.

adaptfilt.apru

Input Argument	Description
states	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 + projectord - 2).
errstates	Vector of the adaptive filter error states. errstates defaults to a zero vector with length equal to (projectord - 1).
epsstates	Vector of the epsilon values of the adaptive filter. epsstates defaults to a vector of zeros with (projectord - 1) elements.

Properties
Since your adaptfilt.apru filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.apru objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
ProjectionOrder	1 to as large as needed.	Projection order of the affine projection algorithm. ProjectionOrder defines the size of the input signal covariance matrix and defaults to two.

Name	Range	Description
OffsetCov	Matrix of values	Contains the offset covariance matrix
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length 1 vector, the number of filter coefficients. coeffs defaults to length 1 vector of zeros when you do not provide the argument for input.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 + projectord - 2).
ErrorStates	Vector of elements	Vector of the adaptive filter error states. errstates defaults to a zero vector with length equal to (projectord - 1).
EpsilonStates	Vector of elements	Vector of the epsilon values of the adaptive filter. epsstates defaults to a vector of zeros with (projectord - 1) elements.

Name	Range	Description
StepSize	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to true.

Examples   Demonstrate quadrature phase shift keying (QPSK) adaptive equalization using a 32-coefficient FIR filter. This example runs the adaptation process for 1000 iterations.

```
D  = 16;                       % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];     % Numerator coefficients of channel
a  = [1 -0.7];                 % Denominator coefficients of channel
ntr= 1000;                     % Number of iterations
s  = sign(randn(1,ntr+D)) + 1j*sign(randn(1,ntr+D)); % Baseband
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));  % Noise signal
r  = filter(b,a,s)+n;     % Received signal
x  = r(1+D:ntr+D); % Input signal (received signal)
d  = s(1:ntr);     % Desired signal (delayed QPSK signal)
mu = 0.1;                      % Step size
po = 4;                        % Projection order
offset = 0.05;                 % Offset
ha = adaptfilt.apru(32,mu,po,offset); [y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e])); title('In-Phase Components');
```

```
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e])); title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot');
axis('square');  xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

In the following component and scatter plots, you see the results of QPSK equalization.

References   [1] Ozeki. K., T. Omeda, "An Adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and Its Properties," Electronics and Communications in Japan, vol. 67-A, no. 5, pp. 19-27, May 1984

Purpose   FIR adaptive filter that uses block affine projection

Syntax   ha = adaptfilt.bap(l,step,projectord,offset,coeffs,states)

Description   ha = adaptfilt.bap(l,step,projectord,offset,coeffs,states)
                   constructs a block affine projection FIR adaptive filter ha.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.bap.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. l defaults to 10.
step	Affine projection step size. This is a scalar that should be a value between zero and one. Setting step equal to one provides the fastest convergence during adaptation. step defaults to 1.
projectord	Projection order of the affine projection algorithm. projectord defines the size of the input signal covariance matrix and defaults to two.
offset	Offset for the input signal covariance matrix. You should initialize the covariance matrix to a diagonal matrix whose diagonal entries are equal to the offset you specify. offset should be positive. offset defaults to one.

Input Argument	Description
coeffs	Vector containing the initial filter coefficients. It must be a length l vector, the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
states	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).

Properties   Since your adaptfilt.bap filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.bap objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
ProjectionOrder	1 to as large as needed.	Projection order of the affine projection algorithm. ProjectionOrder defines the size of the input signal covariance matrix and defaults to two.
OffsetCov	Matrix of values	Contains the offset covariance matrix

Name	Range	Description
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector, the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).
StepSize	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to true.

Examples   Show an example of quadrature phase shift keying (QPSK) adaptive equalization using a 32-coefficient FIR filter.

```
D = 16;                          % delay
```

```
b  = exp(1j*pi/4)*[-0.7 1];                % Numerator coefficients
a  = [1 -0.7];                             % Denominator coefficients
ntr= 1000;                                 % Number of iterations
s  = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D));% Baseband signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));    % Noise signal
r  = filter(b,a,s)+n;                      % Received signal
x  = r(1+D:ntr+D);                         % Input signal (received signal)
d  = s(1:ntr);                             % Desired signal (delayed QPSK signal)
mu = 0.5;                                  % Step size
po = 4;                                    % Projection order
offset = 1.0;                              % Offset for covariance matrix
ha = adaptfilt.bap(32,mu,po,offset);
[y,e] = filter(ha,x,d); subplot(2,2,1);
plot(1:ntr,real([d;y;e]));
title('In-Phase Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

Using the block affine projection object in QPSK results in the plots
shown here.

References   [1] Ozeki, K. and T. Omeda, "An Adaptive Filtering Algorithm Using an Orthogonal Projection to an Affine Subspace and Its Properties," Electronics and Communications in Japan, vol. 67-A, no. 5, pp. 19-27, May 1984

[2] Montazeri, M. and Duhamel, P, "A Set of Algorithms Linking NLMS and Block RLS Algorithms," IEEE Transactions Signal Processing, vol. 43, no. 2, pp, 444-453, February 1995

See Also   adaptfilt | adaptfilt.ap | adaptfilt.apru

adaptfilt.blms

Purpose   FIR adaptive filter that uses BLMS

Syntax   ha = adaptfilt.blms(l,step,leakage,blocklen,coeffs,states)

Description   ha = adaptfilt.blms(l,step,leakage,blocklen,coeffs,states)
constructs an FIR block LMS adaptive filter ha, where l is the adaptive
filter length (the number of coefficients or taps) and must be a positive
integer. l defaults to 10.

step is the block LMS step size. You must set step to a nonnegative
scalar. You can use function maxstep to determine a reasonable range
of step size values for the signals being processed. When unspecified,
step defaults to 0.

leakage is the block LMS leakage factor. It must be a scalar between 0
and 1. If you set leakage to be less than one, you implement the leaky
block LMS algorithm. leakage defaults to 1 specifying no leakage in
the adapting algorithm.

blocklen is the block length used. It must be a positive integer and
the signal vectors d and x should be divisible by blocklen. Larger
block lengths result in faster per-sample execution times but with
poor adaptation characteristics. When you choose blocklen such that
blocklen + length(coeffs) is a power of 2, use adaptfilt.blmsfft.
blocklen defaults to l.

coeffs is a vector of initial filter coefficients. it must be a length l
vector. coeffs defaults to length l vector of zeros.

states contains a vector of your initial filter states. It must be a length
l vector and defaults to a length l vector of zeros when you do not
include it in your calling function.

Properties   In the syntax for creating the adaptfilt object, the input options are
properties of the object created. This table lists the properties for the
adjoint LMS object, their default values, and a brief description of the
property.

Property	Default Value	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
States	Vector of elements	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to l
Leakage		Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
BlockLength	Vector of length l	Size of the blocks of data processed in each iteration

Property	Default Value	Description
StepSize	0.1	Sets the block LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution. Use maxstep to determine the maximum usable step size.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

Examples   Use an adaptive filter to identify an unknown 32nd-order FIR filter. In this example 500 input samples result in 500 iterations of the adaptation process. You see in the plot that follows the example code that the adaptive filter has determined the coefficients of the unknown system under test.

```
x  = randn(1,500);        % Input to the filter
b  = fir1(31,0.5);        % FIR system to be identified
no  = 0.1*randn(1,500);   % Observation noise signal
d  = filter(b,1,x)+no;    % Desired signal
mu = 0.008;               % Block LMS step size
n  = 5;                   % Block length
ha = adaptfilt.blms(32,mu,1,n);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
```

```
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');
grid on;
```

Based on looking at the figures here, the adaptive filter correctly identified the unknown system after 500 iterations, or fewer. In the lower plot, you see the comparison between the actual filter coefficients and those determined by the adaptation process.

References   Shynk, J.J.,"Frequency-Domain and Multirate Adaptive Filtering," IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

See Also   adaptfilt.blmsfft | adaptfilt.fdaf | adaptfilt.lms

adaptfilt.blmsfft

Purpose

FIR adaptive filter that uses FFT-based BLMS

Syntax

```
ha = adaptfilt.blmsfft(l,step,leakage,blocklen,coeffs,
states)
```

Description

`ha = adaptfilt.blmsfft(l,step,leakage,blocklen,coeffs, states)` constructs an FIR block LMS adaptive filter object `ha` where `l` is the adaptive filter length (the number of coefficients or taps) and must be a positive integer. `l` defaults to 10. `step` is the block LMS step size. It must be a nonnegative scalar. The function `maxstep` may be helpful to determine a reasonable range of step size values for the signals you are processing. `step` defaults to 0.

`leakage` is the block LMS leakage factor. It must also be a scalar between 0 and 1. When `leakage` is less than one, the `adaptfilt.blmsfft` implements a leaky block LMS algorithm. `leakage` defaults to 1 (no leakage). `blocklen` is the block length used. It must be a positive integer such that

```
 blocklen + length(coeffs)
```

is a power of two; otherwise, an `adaptfilt.blms` algorithm is used for adapting. Larger block lengths result in faster execution times, with poor adaptation characteristics as the cost of the speed gained. `blocklen` defaults to `l`. Enter your initial filter coefficients in `coeffs`, a vector of length `l`. When omitted, `coeffs` defaults to a length `l` vector of all zeros. `states` contains a vector of initial filter states; it must be a length `l` vector. `states` defaults to a length `l` vector of all zeros when you omit the `states` argument in the calling syntax.

Properties

In the syntax for creating the `adaptfilt` object, the input options are properties of the object you create. This table lists the properties for the block LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coefficients defaults to length l vector of zeros when you do not provide the argument for input.
States	Vector of elements of length l	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to l
Leakage	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
BlockLength	Vector of length l	Size of the blocks of data processed in each iteration

Property	Default Value	Description
StepSize	0.1	Sets the block LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution. Use maxstep to determine the maximum usable step size.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

Examples   Identify an unknown FIR filter with 32 coefficients using 512 iterations of the adapting algorithm.

```
x  = randn(1,512);      % Input to the filter
b  = fir1(31,0.5);      % FIR system to be identified
no = 0.1*randn(1,512);  % Observation noise signal
d  = filter(b,1,x)+no;  % Desired signal
mu = 0.008;             % Step size
n  = 16;                % Block length
ha = adaptfilt.blmsfft(32,mu,1,n);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d(1:500);y(1:500);e(1:500)]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error'); xlabel('Time Index');
```

```
ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('actual','estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

As a result of running the adaptation process, filter object ha now matches the unknown system FIR filter b, based on comparing the filter coefficients derived during adaptation.

References
Shynk, J.J., "Frequency-Domain and Multirate Adaptive Filtering," IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

See Also
adaptfilt.blms | adaptfilt.fdaf | adaptfilt.lms | filter

adaptfilt.dlms

Purpose   FIR adaptive filter that uses delayed LMS

Syntax   `ha = adaptfilt.dlms(l,step,leakage,delay,errstates,coeffs,`
`...states)`

Description   `ha = adaptfilt.dlms(l,step,leakage,delay,errstates,coeffs,`
`...states)` constructs an FIR delayed LMS adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for
`adaptfilt.dlms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	LMS step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.
`leakage`	Your LMS leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.lms` implements a leaky LMS algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`delay`	Update delay given in time samples. This scalar should be a positive integer — negative delays do not work. `delay` defaults to 1.
`errstates`	Vector of the error states of your adaptive filter. It must have a length equal to the update delay (`delay`) in samples. `errstates` defaults to an appropriate length vector of zeros.

Input Argument	Description
coeffs	Vector of initial filter coefficients. it must be a length l vector. coeffs defaults to length l vector with elements equal to zero.
states	Vector of initial filter states for the adaptive filter. It must be a length l-1 vector. states defaults to a length l-1 vector of zeros.

Properties In the syntax for creating the adaptfilt object, the input options are properties of the object you create. This table lists the properties for the block LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input. LMS FIR filter coefficients. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need l entries in coeffs.
Delay	1	Specifies the update delay for the adaptive algorithm.

Property	Default Value	Description
ErrorStates	Vector of zeros with the number of elements equal to delay	A vector comprising the error states for the adaptive filter.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
Leakage	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

Property	Default Value	Description
StepSize	0.1	Sets the LMS algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).

Examples   System identification of a 32-coefficient FIR filter. Refer to the figure that follows to see the results of the adapting filter process.

```
x   = randn(1,500);      % Input to the filter
b   = fir1(31,0.5);      % FIR system to be identified
n   = 0.1*randn(1,500);  % Observation noise signal
d   = filter(b,1,x)+n;   % Desired signal
mu = 0.008;              % LMS step size.
delay = 1;               % Update delay
ha = adaptfilt.dlms(32,mu,1,delay);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

Using a delayed LMS adaptive filter in the process to identify an unknown filter appears to work as planned, as shown in this figure.

adaptfilt.dlms

References   Shynk, J.J.,"Frequency-Domain and Multirate Adaptive Filtering,"
IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

See Also   adaptfilt.adjlms | adaptfilt.filtxlms | adaptfilt.lms

Purpose	FIR adaptive filter that uses frequency-domain with bin step size normalization
Syntax	ha = adaptfilt.fdaf(l,step,leakage,delta,lambda,blocklen, offset,...coeffs,states)
Description	ha = adaptfilt.fdaf(l,step,leakage,delta,lambda,blocklen, offset,...coeffs,states) constructs a frequency-domain FIR adaptive filter ha with bin step size normalization. If you omit all the input arguments you create a default object with l = 10 and step = 1.

Input Arguments

Entries in the following table describe the input arguments for adaptfilt.fdaf.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps). l must be a positive integer; it defaults to 10 when you omit the argument.
step	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. step defaults to 1.
leakage	Leakage parameter of the adaptive filter. If this parameter is set to a value between zero and one, you implement a leaky FDAF algorithm. leakage defaults to 1 — no leakage provided in the algorithm.
delta	Initial common value of all of the FFT input signal powers. Its initial value should be positive. delta defaults to 1.
lambda	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. lambda should lie in the range (0,1]. lambda defaults to 0.9.

Input Argument	Description
blocklen	Block length for the coefficient updates. This must be a positive integer. For faster execution, (blocklen + l) should be a power of two. blocklen defaults to l.
offset	Offset for the normalization terms in the coefficient updates. Use this to avoid divide by zeros or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.
coeffs	Initial time-domain coefficients of the adaptive filter. coeff should be a length l vector. The adaptive filter object uses these coefficients to compute the initial frequency-domain filter coefficients via an FFT computed after zero-padding the time-domain vector by the blocklen.
states	The adaptive filter states. states defaults to a zero vector that has length equal to l.

Properties   Since your adaptfilt.fdaf filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.fdaf objects. To show you the properties that apply, this table lists and describes each property for the adaptfilt.fdaf filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
AvgFactor	(0, 1]	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. Same as the input argument `lambda`.
BlockLength	Any integer	Block length for the coefficient updates. This must be a positive integer. For faster execution, (blocklen + l) should be a power of two. blocklen defaults to l.
FFTCoefficients		Stores the discrete Fourier transform of the filter coefficients in `coeffs`.
FFTStates		States for the FFT operation.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
Leakage		Leakage parameter of the adaptive filter. if this parameter is set to a value between zero and one, you implement a leaky FDAF algorithm. leakage defaults to 1 — no leakage provided in the algorithm.

Name	Range	Description
Offset	Any positive real value	Offset for the normalization terms in the coefficient updates. Use this to avoid dividing by zero or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
Power		A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
StepSize	Any scalar from zero to one, inclusive	Specifies the step size taken between filter coefficient updates

Examples   Quadrature Phase Shift Keying (QPSK) adaptive equalization using 1024 iterations of a 32-coefficient FIR filter. After this example code, a figure demonstrates the equalization results.

```
D = 16;                          % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                   % Denominator coefficients of channel
ntr= 1024;                       % Number of iterations
s  = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D));  %QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));      % Noise signal
r  = filter(b,a,s)+n;            % Received signal
x  = r(1+D:ntr+D);               % Input signal (received signal)
d  = s(1:ntr);                   % Desired signal (delayed QPSK signal)
del = 1;                         % Initial FFT input powers
mu  = 0.1;                       % Step size
lam = 0.9;                       % Averaging factor
ha = adaptfilt.fdaf(32,mu,1,del,lam);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e])); title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e])); title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

References   Shynk, J.J.,"Frequency-Domain and Multirate Adaptive Filtering,"
IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992

See Also   `adaptfilt.ufdaf` | `adaptfilt.pbfdaf` | `adaptfilt.blms` |
`adaptfilt.blmsfft`

Purpose	FIR adaptive filter that uses filtered-x LMS
Syntax	`ha = adaptfilt.filtxlms(l,step,leakage,pathcoeffs,` `pathest,...errstates,pstates,coeffs,states)`
Description	`ha = adaptfilt.filtxlms(l,step,leakage,pathcoeffs,` `pathest,...errstates,pstates,coeffs,states)` constructs an filtered-x LMS adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.filtxlms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Filtered LMS step size. it must be a nonnegative scalar. `step` defaults to 0.1.
`leakage`	is the filtered-x LMS leakage factor. it must be a scalar between 0 and 1. If it is less than one, a leaky version of `adaptfilt.filtxlms` is implemented. `leakage` defaults to 1 (no leakage).
`pathcoeffs`	is the secondary path filter model. this vector should contain the coefficient values of the secondary path from the output actuator to the error sensor.
`pathest`	is the estimate of the secondary path filter model. `pathest` defaults to the values in `pathcoeffs`.
`fstates`	is a vector of filtered input states of the adaptive filter. `fstates` defaults to a zero vector of length equal to (`l` - 1).

Input Argument	Description
pstates	are the secondary path FIR filter states. it must be a vector of length equal to the (length(pathcoeffs) - 1). pstates defaults to a vector of zeros of appropriate length.
coeffs	is a vector of initial filter coefficients. it must be a length l vector. coeffs defaults to length l vector of zeros.
states	Vector of initial filter states. states defaults to a zero vector of length equal to the larger of (length(pathcoeffs) - 1) and (length(pathest) - 1).

Properties   In the syntax for creating the adaptfilt object, the input options are properties of the object created. This table lists the properties for the adjoint LMS object, their default values, and a brief description of the property.

Property	Default Value	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.

Property	Default Value	Description
FilteredInputStates	l-1	Vector of filtered input states with length equal to l - 1.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
States	Vector of elements	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2)
SecondaryPathCoeffs	No default	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor
SecondaryPathEstimate	pathcoeffs values	An estimate of the secondary path filter model
SecondaryPathStates	Vector of size (length (pathcoeffs) -1) with all elements equal to zero.	The states of the secondary path FIR filter — the unknown system
StepSize	0.1	Sets the filtered-x algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the

Property	Default Value	Description
		adaptive filter converges to the filter solution.

Examples   Demonstrate active noise control of a random noise signal over 1000 iterations.

As the figure that follows this code demonstrates, the filtered-x LMS filter successfully controls random noise in this context.

```
x  = randn(1,1000);      % Noise source
g  = fir1(47,0.4);       % FIR primary path system model
n  = 0.1*randn(1,1000);  % Observation noise signal
d  = filter(g,1,x)+n;    % Signal to be cancelled
b  = fir1(31,0.5);       % FIR secondary path system model
mu = 0.008;              % Filtered-X LMS step size
ha = adaptfilt.filtxlms(32,mu,1,b);
[y,e] = filter(ha,x,d); plot(1:1000,d,'b',1:1000,e,'r');
title('Active Noise Control of a Random Noise Signal');
legend('Original','Attenuated');
xlabel('Time Index'); ylabel('Signal Value');  grid on;
```

See also   adaptfilt.dlms, adaptfilt.lms

References   Kuo, S.M., and Morgan, D.R. *Active Noise Control Systems: Algorithms and DSP Implementations*, New York, N.Y: John Wiley & Sons, 1996.

Widrow, B., and Stearns, S.D. *Adaptive Signal Processing*, Upper Saddle River, N.J: Prentice Hall, 1985.

Purpose   Fast transversal LMS adaptive filter

Syntax   `ha = adaptfilt.ftf(l,lambda,delta,gamma,gstates,coeffs,`
`states)`

Description   `ha = adaptfilt.ftf(l,lambda,delta,gamma,gstates,coeffs,`
`states)` constructs a fast transversal least squares adaptive filter
object `ha`.

Input Arguments

Entries in the following table describe the input arguments for
`adaptfilt.ftf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`lambda`	RLS forgetting factor. This is a scalar that should lie in the range (1-0.5/l, 1]. `lambda` defaults to 1.
`delta`	Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. `delta` defaults to one.
`gamma`	Conversion factor. `gamma` defaults to one specifying soft-constrained initialization.
`gstates`	States of the Kalman gain updates. `gstates` defaults to a zero vector of length `l`.
`coeffs`	Length `l` vector of initial filter coefficients. `coeffs` defaults to a length `l` vector of zeros.
`states`	Vector of initial filter States. `states` defaults to a zero vector of length (l-1).

Properties

Since your `adaptfilt.ftf` filter is an object, it has properties that define its operating behavior. Note that many of the properties are also input arguments for creating `adaptfilt.ftf` objects. To show you the properties that apply, this table lists and describes each property for the fast transversal least squares filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
BkwdPrediction		Returns the predicted samples generated during adaptation. Refer to [2] in the bibliography for details about linear prediction.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
ConversionFactor		Conversion factor. Called gamma when it is an input argument, it defaults to the matrix [1 -1] that specifies soft-constrained initialization.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
ForgettingFactor		RLS forgetting factor. This is a scalar that should lie in the range (1-0.5/l, 1]. lambda defaults to 1.

Name	Range	Description
FwdPrediction		Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
InitFactor		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. delta defaults to one.
KalmanGain		Empty when you construct the object, this gets populated after you run the filter.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).

Examples   System Identification of a 32-coefficient FIR filter by running the identification process for 500 iterations.

```
x   = randn(1,500);    % Input to the filter
b   = fir1(31,0.5);    % FIR system to be identified
n   = 0.1*randn(1,500);% Observation noise signal
d   = filter(b,1,x)+n; % Desired signal
N   = 31;              % Adaptive filter order
lam = 0.99;            % RLS forgetting factor
del = 0.1;             % Soft-constrained initialization factor
ha = adaptfilt.ftf(32,lam,del);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated'); grid on;
xlabel('coefficient #'); ylabel('Coefficient Value');
```

For this example of identifying an unknown system, the figure shows that the adaptation process identifies the filter coefficients for the unknown FIR filter within the first 150 iterations.

adaptfilt.ftf

References   D.T.M. Slock and Kailath, T., "Numerically Stable Fast Transversal Filters for Recursive Least Squares Adaptive Filtering," IEEE Trans. Signal Processing, vol. 38, no. 1, pp. 92-114.

See Also   `adaptfilt.swftf` | `adaptfilt.rls` | `adaptfilt.lsl`

Purpose   FIR adaptive filter that uses gradient lattice

Syntax   ha = adaptfilt.gal(l,step,leakage,offset,rstep,delta,
lambda,...rcoeffs,coeffs,states)

Description   ha = adaptfilt.gal(l,step,leakage,offset,rstep,delta,
lambda,...rcoeffs,coeffs,states) constructs a gradient adaptive
lattice FIR filter ha.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.gal.

Input Argument	Description
l	Length of the joint process filter coefficients. It must be a positive integer and must be equal to the length of the reflection coefficients plus one. l defaults to 10.
step	Joint process step size of the adaptive filter. This scalar should be a value between zero and one. step defaults to 0.
leakage	Leakage factor of the adaptive filter. It must be a scalar between 0 and 1. Setting leakage less than one implements a leaky algorithm to estimate both the reflection and the joint process coefficients. leakage defaults to 1 (no leakage).
offset	Specifies an optional offset for the denominator of the step size normalization term. It must be a scalar greater or equal to zero. A non-zero offset is useful to avoid divide-by-near-zero conditions when the input signal amplitude becomes very small. offset defaults to 1.

Input Argument	Description
rstep	Reflection process step size of the adaptive filter. This scalar should be a value between zero and one. rstep defaults to step.
delta	Initial common value of the forward and backward prediction error powers. It should be a positive value. 0.1 is the default value for delta.
lambda	Specifies the averaging factor used to compute the exponentially windowed forward and backward prediction error powers for the coefficient updates. lambda should lie in the range (0, 1]. lambda defaults to the value (1 - step).
rcoeffs	Vector of initial reflection coefficients. It should be a length (l-1) vector. rcoeffs defaults to a zero vector of length (l-1).
coeffs	Vector of initial joint process filter coefficients. It must be a length l vector. coeffs defaults to a length l vector of zeros.
states	Vector of the backward prediction error states of the adaptive filter. states defaults to a zero vector of length (l-1).

Properties   Since your adaptfilt.gal filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.gal objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
AvgFactor		Specifies the averaging factor used to compute the exponentially-windowed forward and backward prediction error powers for the coefficient updates. Same as the input argument lambda.
BkwdPredErrorPower		Returns the minimum mean-squared prediction error. Refer to [2] in the bibliography for details about linear prediction
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
FwdPredErrorPower		Returns the minimum mean-squared prediction error in the forward direction. Refer to [2] in the bibliography for details about linear prediction.

Name	Range	Description
Leakage	0 to 1	Leakage parameter of the adaptive filter. If this parameter is set to a value between zero and one, you implement a leaky GAL algorithm. leakage defaults to 1 — no leakage provided in the algorithm.
Offset		Offset for the normalization terms in the coefficient updates. Use this to avoid dividing by zero or by very small numbers when input signal amplitude becomes very small. offset defaults to one.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
ReflectionCoeffs		Coefficients determined for the reflection portion of the filter during adaptation.

Name	Range	Description
ReflectionCoeffsStep		Size of the steps used to determine the reflection coefficients.
States	Vector of elements	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 + projectord - 2).
StepSize	0 to 1	Specifies the step size taken between filter coefficient updates

Examples

Perform a Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter over 1000 iterations.

```
D  = 16;                    % Number of delay samples
b  = exp(1j*pi/4)*[-0.7 1];  % Numerator coefficients
a  = [1 -0.7];              % Denominator coefficients
ntr= 1000;                  % Number of iterations
s  = sign(randn(1,ntr+D)) + 1j*sign(randn(1,ntr+D)); % QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));    % Noise signal
r  = filter(b,a,s)+n;       % Received signal
x  = r(1+D:ntr+D);          % Input signal (received signal)
d  = s(1:ntr);              % Desired signal (delayed QPSK signal)
L = 32;                     % filter length
mu = 0.007;                 % Step size
ha = adaptfilt.gal(L,mu);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('signal value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
```

```
subplot(2,2,3); plot(x(ntr-100:ntr),'.');  axis([-3 3 -3 3]);
title('Received Signal Scatter Plot');  axis('square');
xlabel('Real[x]'); ylabel('Imag[x]');  grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');
axis([-3 3 -3 3]); title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');  grid on;
```

To see the results, look at this figure.

References   Griffiths, L.J. "A Continuously Adaptive Filter Implemented as a Lattice Structure," Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Hartford, CT, pp. 683-686, 1977

Haykin, S.,*Adaptive Filter Theory*, 3rd Ed., Upper Saddle River, NJ, Prentice Hall, 1996

See Also   adaptfilt.qrdlsl | adaptfilt.lsl | adaptfilt.tdafdft

Purpose	FIR adaptive filter that uses householder (RLS)
Syntax	ha = adaptfilt.hrls(l,lambda,sqrtinvcov,coeffs,states)

Description   ha = adaptfilt.hrls(l,lambda,sqrtinvcov,coeffs,states)
constructs an FIR householder RLS adaptive filter ha.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.hrls.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. l defaults to 10.
lambda	RLS forgetting factor. This is a scalar and should lie in the range (0, 1]. lambda defaults to 1 meaning the adaptation process retains infinite memory.
sqrtinvcov	Square-root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
coeffs	Vector of initial filter coefficients. It must be a length l vector. coeffs defaults to being a length l vector of zeros.
states	Vector of initial filter states. It must be a length l-1 vector. states defaults to a length l-1 vector of zeros.

Properties   Since your adaptfilt.hrls filter is an object, it has properties that
define its behavior in operation. Note that many of the properties are
also input arguments for creating adaptfilt.hrls objects. To show you
the properties that apply, this table lists and describes each property
for the affine projection filter object.

adaptfilt.hrls

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
ForgettingFactor	Scalar	RLS forgetting factor. This is a scalar and should lie in the range (0, 1]. Same as input argument lambda. It defaults to 1 meaning the adaptation process retains infinite memory.
KalmanGain	Vector of size (l,1)	Empty when you construct the object, this gets populated after you run the filter.

Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. Defaults to false.
SqrtInvCov	Matrix of doubles	Square root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l - 1).

Examples   Use 500 iterations of an adaptive filter object to identify a 32-coefficient FIR filter system. Both the example code and the resulting figure show the successful filter identification through adaptive filter processing.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
G0 = sqrt(10)*eye(32); % Initial sqrt correlation matrix inverse
lam = 0.99;            % RLS forgetting factor
ha = adaptfilt.hrls(32,lam,G0);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
```

```
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated'); grid on;
 xlabel('Coefficient #'); ylabel('Coefficient Value');
```


See Also   adaptfilt.hswrls | adaptfilt.qrdrls | adaptfilt.rls

Purpose   FIR adaptive filter that uses householder sliding window RLS

Syntax   ha = adaptfilt.hswrls(l,lambda,sqrtinvcov,swblocklen,
                 dstates,coeffs,states)

Description ha = adaptfilt.hswrls(l,lambda,sqrtinvcov,swblocklen,
                 dstates,coeffs,states) constructs an FIR householder sliding
                 window recursive-least-square adaptive filter ha.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.hswrls.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. l defaults to 10.
lambda	Recursive least square (RLS) forgetting factor. This is a scalar and should lie in the range (0, 1]. lambda defaults to 1 meaning the adaptation process retains infinite memory.
sqrtinvcov	Square-root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
swblocklen	Block length of the sliding window. This integer must be at least as large as the filter length. swblocklen defaults to 16.
dstates	Desired signal states of the adaptive filter. dstates defaults to a zero vector with length equal to (swblocklen - 1).

Input Argument	Description
coeffs	Vector of initial filter coefficients. It must be a length l vector. coeffs defaults to being a length l vector of zeros.
states	Vector of initial filter states. It must be a length (l + swblocklen - 2) vector. states defaults to a length (l + swblocklen -2) vector of zeros.

Properties

Since your adaptfilt.hswrls filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.hswrls objects. To show you the properties that apply, this table lists and describes each property for the affine projection filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
DesiredSignalStates	Vector	Desired signal states of the adaptive filter. dstates defaults to a zero vector with length equal to (swblocklen - 1).

Name	Range	Description
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
ForgettingFactor	Scalar	Root-least-square (RLS) forgetting factor. This is a scalar and should lie in the range (0, 1]. Same as input argument `lambda`. It defaults to 1 meaning the adaptation process retains infinite memory.
KalmanGain	(1,1) vector	Empty when you construct the object, this gets populated after you run the filter.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. Defaults to `false`.
SqrtInvCov	1-by-1 Matrix	Square-root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.

Name	Range	Description
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).
SwBlockLength	Integer	Block length of the sliding window. This integer must be at least as large as the filter length. swblocklen defaults to 16.

Examples   System Identification of a 32-coefficient FIR filter.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
G0 = sqrt(10)*eye(32); % Initial sqrt correlation matrix inverse
lam = 0.99;            % RLS forgetting factor
N  = 64;               % block length
ha = adaptfilt.hswrls(32,lam,G0,N);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

In the pair of plots shown in the figure you see the comparison of the desired and actual output for the adapting filter and the coefficients of both filters, the unknown and the adapted.

See Also   adaptfilt.hrls | adaptfilt.qrdrls | adaptfilt.rls

adaptfilt.lms

Purpose   FIR adaptive filter that uses LMS

Syntax   `ha = adaptfilt.lms(l,step,leakage,coeffs,states)`

Description   `ha = adaptfilt.lms(l,step,leakage,coeffs,states)` constructs an FIR LMS adaptive filter object `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.lms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	LMS step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.1.
`leakage`	Your LMS leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.lms` implements a leaky LMS algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length l-1 vector. `states` defaults to a length l-1 vector of zeros.

Properties   In the syntax for creating the adaptfilt object, the input options are properties of the object created. This table lists the properties for the adaptfilt.lms object, their default values, and a brief description of the property.

Property	Range	Property Description
Algorithm	None	Reports the adaptive filter algorithm the object uses during adaptation
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to a length l vector of zeros when you do not provide the vector as an input argument.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
Leakage	0 to 1	LMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. leakage defaults to 1 (no leakage).

Property	Range	Property Description
PersistentMemory	false or true	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to false.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l - 1).
StepSize	0 to 1	LMS step size. It must be a scalar between zero and one. Setting this step size value to one provides the fastest convergence. step defaults to 0.1.

Examples
Use 500 iterations of an adapting filter system to identify and unknown 32nd-order FIR filter.

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 0.008;            % LMS step size.
ha = adaptfilt.lms(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
```

```
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');  grid on;
```

Using LMS filters in an adaptive filter architecture is a time honored means for identifying an unknown filter. By running the example code provided you can demonstrate one process to identify an unknown FIR filter.

References   Shynk J.J., "Frequency-Domain and Multirate Adaptive Filtering," IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992.

See Also   `adaptfilt.blms | adaptfilt.blmsfft | adaptfilt.dlms | adaptfilt.nlms | adaptfilt.tdafdft | adaptfilt.sd | adaptfilt.se | adaptfilt.ss`

adaptfilt.lsl

Purpose
Least squares lattice adaptive filter

Syntax
`ha = adaptfilt.lsl(l,lambda,delta,coeffs,states)`

Description
`ha = adaptfilt.lsl(l,lambda,delta,coeffs,states)` constructs a least squares lattice adaptive filter ha.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.lsl`.

Input Argument	Description
l	Length of the joint process filter coefficients. It must be a positive integer and must be equal to the length of the prediction coefficients plus one. L defaults to 10.
lambda	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. lambda defaults to 1. lambda = 1 denotes infinite memory while adapting to find the new filter.
delta	Soft-constrained initialization factor in the least squares lattice algorithm. It should be positive. delta defaults to 1.
coeffs	Vector of initial joint process filter coefficients. It must be a length l vector. coeffs defaults to a length l vector of all zeros.
states	Vector of the backward prediction error states of the adaptive filter. states defaults to a length l vector of all zeros, specifying soft-constrained initialization for the algorithm.

Properties
Since your `adaptfilt.lsl` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are

also input arguments for creating `adaptfilt.lsl` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
BkwdPrediction		Returns the predicted samples generated during adaptation. Refer to [2] in the bibliography for details about linear prediction.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
ForgettingFactor		Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting forgetting factor = 1 denotes infinite memory while adapting to find the new filter. Note that this is the lambda input argument.

Name	Range	Description
FwdPrediction		Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
InitFactor		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. delta defaults to one.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to l.

Examples   Demonstrate Quadrature Phase Shift Keying (QPSK) adaptive
equalization using a 32-coefficient adaptive filter running for 1000
iterations. After you review the example code, the figure shows the
results of running the example to use QPSK adaptive equalization with
a 32nd-order FIR filter. The error between the in-phase and quadrature
components, as shown by the errors plotted in the upper plots, falls to
near zero. Also, the equalized signal shows the clear quadrature nature.

```
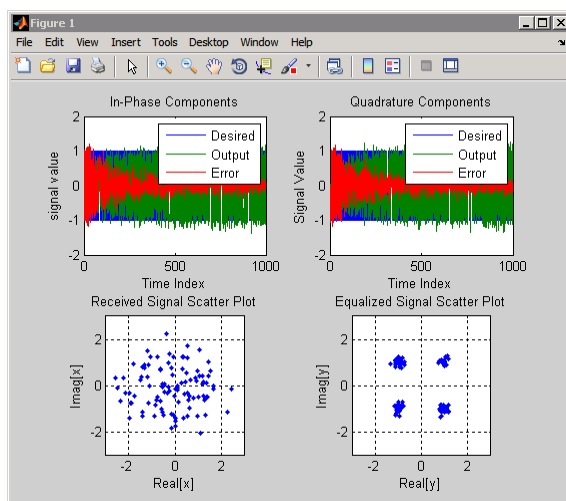D  = 16;                       % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];    % Numerator coefficients of channel
a  = [1 -0.7];                 % Denominator coefficients of channel
ntr= 1000;                     % Number of iterations
s  = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D)); % QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));     % Noise signal
r  = filter(b,a,s)+n;          % Received signal
x  = r(1+D:ntr+D);             % Input signal (received signal)
d  = s(1:ntr);                 % Desired signal (delayed QPSK signal)
lam = 0.995;                   % Forgetting factor
del = 1;                       % initialization factor
ha = adaptfilt.lsl(32,lam,del);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');  axis([-3 3 -3 3]);
title('Received Signal Scatter Plot');  axis('square');
xlabel('Real[x]'); ylabel('Imag[x]');  grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');  axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');  grid on;
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');
```

adaptfilt.lsl

References   Haykin, S., *Adaptive Filter Theory*, 2nd Edition, Prentice Hall, N.J.,
1991

See Also   adaptfilt.qrdlsl | adaptfilt.gal | adaptfilt.ftf |
adaptfilt.rls

Purpose   Normalized least mean squares adaptive filter

Syntax   `ha = adaptfilt.nlms(l,step,leakage,offset,coeffs,states)`

Description `ha = adaptfilt.nlms(l,step,leakage,offset,coeffs,states)`
constructs a normalized least-mean squares (NLMS) FIR adaptive filter
object named `ha`.

Input Arguments

Entries in the following table describe the input arguments for
`adaptfilt.nlms`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	NLMS step size. It must be a scalar between 0 and 2. Setting this step size value to one provides the fastest convergence. `step` defaults to 1.
`leakage`	NLMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. `leakage` defaults to 1 (no leakage).
`offset`	Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors. Use this to avoid dividing by zero (or by very small numbers) when the square of the input data norm becomes very small (when the input signal amplitude becomes very small). When you omit it, `offset` defaults to zero.

Input Argument	Description
coeffs	Vector composed of your initial filter coefficients. Enter a length l vector. coeffs defaults to a vector of zeros with length equal to the filter order.
states	Your initial adaptive filter states appear in the states vector. It must be a vector of length l-1. states defaults to a length l-1 vector with zeros for all of the elements.

Properties

In the syntax for creating the adaptfilt object, the input options are properties of the object you create. This table lists the properties for normalized LMS objects, their default values, and a brief description of the property.

Property	Range	Property Description
Algorithm	None	Reports the adaptive filter algorithm the object uses during adaptation
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps

Property	Range	Property Description
Leakage	0 to 1	NLMS leakage factor. It must be a scalar between zero and one. When it is less than one, a leaky NLMS algorithm results. leakage defaults to 1 (no leakage).
Offset	0 or greater	Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors. Use this to avoid dividing by zero (or by very small numbers) when the square of the input data norm becomes very small (when the input signal amplitude becomes very small). When you omit it, offset defaults to zero.
PersistentMemory	false or true	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to false.

Property	Range	Property Description
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l - 1).
StepSize	0 to 1	NLMS step size. It must be a scalar between zero and one. Setting this step size value to one provides the fastest convergence. step defaults to one.

Examples   To help you compare this algorithm's performance to other LMS-based algorithms, such as BLMS or LMS, this example demonstrates the NLMS adaptive filter in use to identify the coefficients of an unknown FIR filter of order equal to 32 — an example used in other adaptive filter examples.

```
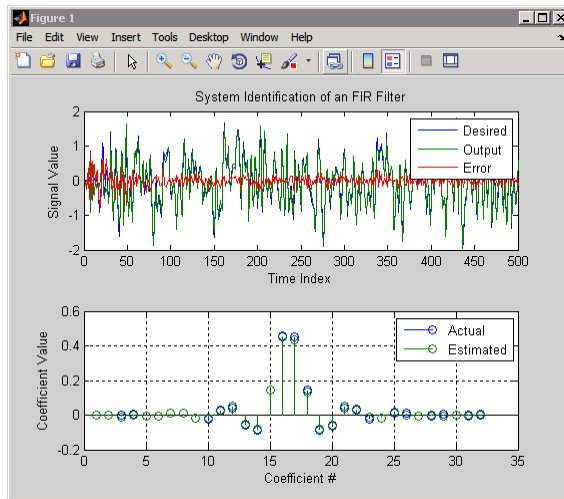x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 1;                % NLMS step size
offset = 50;           % NLMS offset
ha = adaptfilt.nlms(32,mu,1,offset);
[y,e] = filter(ha,x,d);
subplot(2,1,1);
plot(1:500,[d;y;e]);
legend('Desired','Output','Error');
title('System Identification of FIR Filter');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2);
stem([b', ha.coefficients']);
legend('Actual','Estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

As you see from the figure, the nlms variant again closely matches the actual filter coefficients in the unknown FIR filter.

See Also   adaptfilt.ap | adaptfilt.apru | adaptfilt.lms | adaptfilt.rls | adaptfilt.swrls

adaptfilt.pbfdaf

Purpose   Partitioned block frequency-domain FIR adaptive filter

Syntax   `ha = adaptfilt.pbfdaf(l,step,leakage,delta,lambda,blocklen,`
                                `offset,coeffs,states)`

Description   `ha = adaptfilt.pbfdaf(l,step,leakage,delta,lambda,blocklen,`
`offset,coeffs,states)` constructs a partitioned block
frequency-domain FIR adaptive filter `ha` that uses bin step size
normalization during adaptation.

Input Arguments

Entries in the following table describe the input arguments for
`adaptfilt.pbfdaf`.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. L defaults to 10.
step	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. `step` defaults to 1.
leakage	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. `leakage` defaults to 1— no leakage.
delta	Initial common value of all of the FFT input signal powers. Its initial value should be positive. `delta` defaults to 1.
lambda	Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. `lambda` should lie in the range (0,1]. `lambda` defaults to 0.9.

Input Argument	Description
blocklen	Block length for the coefficient updates. This must be a positive integer such that (l/blocklen) is also an integer. For faster execution, blocklen should be a power of two. blocklen defaults to two.
offset	Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small. offset defaults to zero.
coeffs	Initial time-domain coefficients of the adaptive filter. It should be a vector of length l. The PBFDAF algorithm uses these coefficients to compute the initial frequency-domain filter coefficient matrix via FFTs.
states	Specifies the filter initial conditions. states defaults to a zero vector of length l.

Properties   Since your adaptfilt.pbfdaf filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.pbfdaf objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
AvgFactor		Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. AvgFactor should lie in the range (0,1]. AvgFactor defaults to 0.9. Called lambda as an input argument.
BlockLength		Block length for the coefficient updates. This must be a positive integer such that (1/blocklen) is also an integer. For faster execution, blocklen should be a power of two. blocklen defaults to two.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
FFTCoefficients		Stores the discrete Fourier transform of the filter coefficients in coeffs.
FFTStates		States for the FFT operation.

Name	Range	Description
Leakage	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. leakage defaults to 1 — no leakage.
Offset		Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small. offset defaults to zero.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

Name	Range	Description
Power		A vector of 2*1 elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
StepSize	0 to 1	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. step defaults to 1.

Examples

An example of Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter.

```
D   = 16;                         % Number of samples of delay
b   = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a   = [1 -0.7];                   % Denominator coefficients of channel
ntr = 1000;                       % Number of iterations
s   = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D)); % Baseband QPSK signal
n   = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));     % Noise signal
r   = filter(b,a,s)+n;    % Received signal
x   = r(1+D:ntr+D);               % Input signal (received signal)
d   = s(1:ntr);                   % Desired signal (delayed QPSK signal)
del = 1;                          % Initial FFT input powers
mu  = 0.1;                        % Step size
lam = 0.9;                        % Averaging factor
N   = 8;                          % Block size
ha = adaptfilt.pbfdaf(32,mu,1,del,lam,N);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e])); title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
```

```
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');  axis([-3 3 -3 3]);
 title('Received Signal Scatter Plot');
axis('square');  xlabel('Real[x]'); ylabel('Imag[x]');  grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');  axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');  grid on;subplot(2,2,4); plot(y(ntr
title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');  grid on;
```

In the figure shown, the four subplots provide the details of the results of the QPSK process used in the equalization for this example.

References   So, J.S. and K.K. Pang, "Multidelay Block Frequency Domain Adaptive Filter," IEEE Trans. Acoustics, Speech, and Signal Processing, vol. 38, no. 2, pp. 373-376, February 1990

Paez Borrallo, J.M.and M.G. Otero, "On The Implementation of a Partitioned Block Frequency Domain Adaptive Filter (PBFDAF) For Long Acoustic Echo Cancellation," Signal Processing, vol. 27, no. 3, pp. 301-315, June 1992

adaptfilt.pbfdaf

See Also adaptfilt.fdaf | adaptfilt.pbufdaf | adaptfilt.blmsfft

Purpose   Partitioned block unconstrained frequency-domain adaptive filter

Syntax   ha = adaptfilt.pbufdaf(l,step,leakage,delta,lambda,
blocklen,...offset,coeffs,states)

Description   ha = adaptfilt.pbufdaf(l,step,leakage,delta,lambda,
blocklen,...offset,coeffs,states) constructs a partitioned block
unconstrained frequency-domain FIR adaptive filter ha with bin step
size normalization.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.pbufdaf.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. L defaults to 10.
step	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. step defaults to 1.
leakage	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. leakage defaults to 1 — no leakage.
delta	Initial common value of all of the FFT input signal powers. Its initial value should be positive. delta defaults to 1.
lambda	Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. lambda should lie in the range (0,1]. lambda defaults to 0.9.

Input Argument	Description
blocklen	Block length for the coefficient updates. This must be a positive integer such that (1/blocklen) is also an integer. For faster execution, blocklen should be a power of two. blocklen defaults to two.
offset	Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small. offset defaults to zero.
coeffs	Initial time-domain coefficients of the adaptive filter. It should be a vector of length l. The PBFDAF algorithm uses these coefficients to compute the initial frequency-domain filter coefficient matrix via FFTs.
states	Specifies the filter initial conditions. states defaults to a zero vector of length l.

Properties   Since your adaptfilt.pbufdaf filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.pbufdaf objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
AvgFactor		Averaging factor used to compute the exponentially windowed FFT input signal powers for the coefficient updates. AvgFactor should lie in the range (0,1]. AvgFactor defaults to 0.9. Called lambda as an input argument.
BlockLength		Block length for the coefficient updates. This must be a positive integer such that (l/blocklen) is also an integer. For faster execution, blocklen should be a power of two. blocklen defaults to two.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
FFTCoefficients		Stores the discrete Fourier transform of the filter coefficients in coeffs.
FFTStates		States for the FFT operation.

Name	Range	Description
Leakage	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, a leaky version of the PBFDAF algorithm is implemented. leakage defaults to 1 — no leakage.
Offset		Offset for the normalization terms in the coefficient updates. This can be useful to avoid divide by zeros conditions, or dividing by very small numbers, if any of the FFT input signal powers become very small.voffset defaults to zero.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

Name	Range	Description
Power	2*1 element vector	A vector of 2*1 elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
StepSize	0 to 1	Step size of the adaptive filter. This is a scalar and should lie in the range (0,1]. step defaults to 1.

Examples   Demonstrating Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter. To perform the equalization, this example runs for 1000 iterations.

```
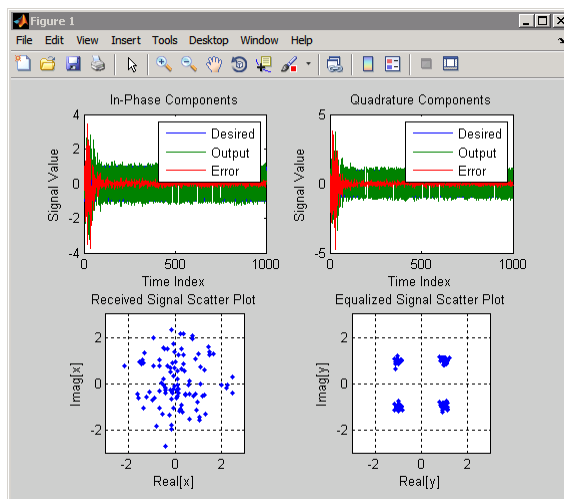D  = 16;       % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1]; % Numerator coefficients of channel
a  = [1 -0.7];              % Denominator coefficients of channel
ntr= 1000;                 % Number of iterations
s  = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D)); % Baseband QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D)); % Noise signal
r  = filter(b,a,s)+n;       % Received signal
x  = r(1+D:ntr+D);          % Input signal (received signal)
d  = s(1:ntr);              % Desired signal (delayed QPSK signal)
del = 1;                    % Initial FFT input powers
mu  = 0.1;                  % Step size
lam = 0.9;                  % Averaging factor
N   = 8;                    % Block size
ha = adaptfilt.pbufdaf(32,mu,1,del,lam,N);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
```

adaptfilt.pbufdaf

```
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

You can compare this algorithm to another, such as the `pbfdaf` version. Use the same example of QPSK adaptation. The following figure shows the results.

References So, J.S. and K.K. Pang, "Multidelay Block Frequency Domain Adaptive Filter," IEEE Trans. Acoustics, Speech, and Signal Processing, vol. 38, no. 2, pp. 373-376, February 1990

Paez Borrallo, J.M. and M.G. Otero, "On The Implementation of a Partitioned Block Frequency Domain Adaptive Filter (PBFDAF) for Long Acoustic Echo Cancellation," Signal Processing, vol. 27, no. 3, pp. 301-315, June 1992

See Also   adaptfilt.ufdaf | adaptfilt.pbfdaf | adaptfilt.blmsfft

adaptfilt.qrdlsl

Purpose	QR-decomposition-based least-squares lattice adaptive filter
Syntax	ha = adaptfilt.qrdlsl(l,lambda,delta,coeffs,states)
Description	ha = adaptfilt.qrdlsl(l,lambda,delta,coeffs,states) returns a QR-decomposition-based least squares lattice adaptive filter ha.

Input Arguments

Entries in the following table describe the input arguments for adaptfilt.qrdlsl.

Input Argument	Description
l	Length of the joint process filter coefficients. It must be a positive integer and must be equal to the length of the prediction coefficients plus one. L defaults to 10.
lambda	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. lambda defaults to 1. lambda = 1 denotes infinite memory while adapting to find the new filter.
delta	Soft-constrained initialization factor in the least squares lattice algorithm. It should be positive. delta defaults to 1.
coeffs	Vector of initial joint process filter coefficients. It must be a length l vector. coeffs defaults to a length l vector of all zeros.
states	Vector of the angle normalized backward prediction error states of the adaptive filter

Properties   Since your adaptfilt.qrdlsl filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.qrdlsl objects. To

show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
BkwdPrediction		Returns the predicted samples generated during adaptation. Refer to [2] in the bibliography for details about linear prediction.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
ForgettingFactor		Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting forgetting factor = 1 denotes infinite memory while adapting to find the new filter. Note that this is the lambda input argument.

adaptfilt.qrdlsl

Name	Range	Description
FwdPrediction		Returns the predicted samples generated during adaptation in the forward direction. Refer to [2] in the bibliography for details about linear prediction.
InitFactor		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. delta defaults to one.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to l -1

Examples   Implement Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter. To see the results of

the equalization process in this example, look at the figure that follows the example code.

```
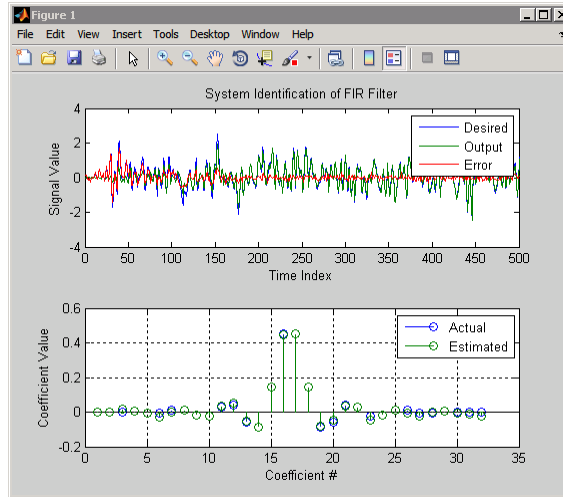D  = 16;                          % Number of samples of delay
b = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                    % Denominator coefficients of channel
ntr= 1000;                        % Number of iterations
s = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D)); % Baseband  QPSK signal
n = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));   % Noise signal
r  = filter(b,a,s)+n;      % Received signal
x = r(1+D:ntr+D);          % Input signal (received signal)
d = s(1:ntr);              % Desired signal (delayed QPSK signal)
lam = 0.995;               % Forgetting factor
del = 1;                   % Soft-constrained initialization factor
ha = adaptfilt.qrdlsl(32,lam,del);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

adaptfilt.qrdlsl

References   Haykin, S.,*Adaptive Filter Theory*, 2nd Edition, Prentice Hall, N.J.,
1991

See Also   `adaptfilt.qrdrls` | `adaptfilt.gal` | `adaptfilt.ftf` |
`adaptfilt.lsl`

Purpose   FIR adaptive filter that uses QR-decomposition-based RLS

Syntax   ha = adaptfilt.qrdrls(l,lambda,sqrtcov,coeffs,states)

Description   ha = adaptfilt.qrdrls(l,lambda,sqrtcov,coeffs,states)
                  constructs an FIR QR-decomposition-based recursive-least squares
                  (RLS) adaptive filter object ha.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.qrdrls.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. l defaults to 10.
lambda	RLS forgetting factor. This is a scalar and should lie within the range (0, 1]. lambda defaults to 1.
sqrtcov	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
coeffs	Vector of initial filter coefficients. It must be a length l vector. coeffs defaults to length l vector whose elements are zeros.
states	Vector of initial filter states. It must be a length l-1 vector. states defaults to a length l-1 vector of zeros.

Properties   Since your adaptfilt.qrdrls filter is an object, it has properties that
                  define its behavior in operation. Note that many of the properties
                  are also input arguments for creating adaptfilt.qrdrls objects. To
                  show you the properties that apply, this table lists and describes each
                  property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	Vector of length l	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
ForgettingFactor	Scalar	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting forgetting factor = 1 denotes infinite memory while adapting to find the new filter. Note that this is the lambda input argument.

Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
SqrtCov	Square matrix with each dimension equal to the filter length l	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
States	Vector of elements	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).

Examples   System Identification of a 32-coefficient FIR filter (500 iterations).

```
x  = randn(1,500);      % Input to the filter
b  = fir1(31,0.5);      % FIR system to be identified
n  = 0.1*randn(1,500);  % Observation noise signal
d  = filter(b,1,x)+n;   % Desired signal
G0 = sqrt(.1)*eye(32);  % Initial sqrt correlation matrix
```

```
lam = 0.99;              % RLS forgetting factor
ha = adaptfilt.qrdrls(32,lam,G0);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated'); grid on;
xlabel('Coefficient #'); ylabel('Coefficient Value');
```

Using this variant of the RLS algorithm successfully identifies the unknown FIR filter, as shown here.

See Also

adaptfilt.rls | adaptfilt.hrls | adaptfilt.hswrls | adaptfilt.swrls

Purpose   Recursive least-squares FIR adaptive filter

Syntax   ha = adaptfilt.rls(l,lambda,invcov,coeffs,states)

Description ha = adaptfilt.rls(l,lambda,invcov,coeffs,states) constructs
                 an FIR direct form RLS adaptive filter ha.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.rls.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. l defaults to 10.
lambda	RLS forgetting factor. This is a scalar and should lie in the range (0, 1]. lambda defaults to 1.
invcov	Inverse of the input signal covariance matrix. For best performance, you should initialize this matrix to be a positive definite matrix.
coeffs	Vector of initial filter coefficients. it must be a length l vector. coeffs defaults to length l vector with elements equal to zero.
states	Vector of initial filter states for the adaptive filter. It must be a length l-1 vector. states defaults to a length l-1 vector of zeros.

Properties Since your adaptfilt.rls filter is an object, it has properties that
                define its behavior in operation. Note that many of the properties are
                also input arguments for creating adaptfilt.rls objects. To show you
                the properties that apply, this table lists and describes each property
                for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
Coefficients	Vector containing l elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps. Remember that filter length is filter order + 1.
ForgettingFactor	Scalar	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting forgetting factor = 1 denotes infinite memory while adapting to find the new filter. Note that this is the lambda input argument.
InvCov	Matrix of size l-by-l	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
KalmanGain	Vector of size (l,1)	Empty when you construct the object, this gets populated after you run the filter.

Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. Defaults to false.
States	Double array	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 + projectord - 2).

Examples   System Identification of a 32-coefficient FIR filter over 500 adaptation iterations.

```
x  = randn(1,500);    % Input to the filter
b  = fir1(31,0.5);    % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n; % Desired signal
P0 = 10*eye(32); % Initial sqrt correlation matrix inverse
lam = 0.99;           % RLS forgetting factor
ha = adaptfilt.rls(32,lam,P0);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient valUe');  grid on;
```

adaptfilt.rls

In this example of adaptive filtering using the RLS algorithm to update the filter coefficients for each iteration, the figure shown reveals the fidelity of the derived filter after adaptation.

See Also   adaptfilt.hrls | adaptfilt.hswrls | adaptfilt.qrdrls

Purpose   Sign-data FIR adaptive filter

Syntax   `ha = adaptfilt.sd(l,step,leakage,coeffs,states)`

Description `ha = adaptfilt.sd(l,step,leakage,coeffs,states)` constructs an
FIR sign-data adaptive filter object `ha`.

Input Arguments

Entries in the following table describe the input arguments for
`adaptfilt.sd`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	SD step size. It must be a nonnegative scalar. `step` defaults to 0.1
`leakage`	Your SD leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.sd` implements a leaky SD algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length `l`-1 vector. `states` defaults to a length `l`-1 vector of zeros.

Properties   In the syntax for creating the `adaptfilt` object, the input options are
properties of the object you create. This table lists the properties for

sign-data objects, their default values, and a brief description of the property.

Property	Default Value	Description
Algorithm	Sign-data	Defines the adaptive filter algorithm the object uses during adaptation.
Coefficients	zeros(1,l)	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input. Should be initialized with the initial coefficients for the FIR filter prior to adapting. You need l entries in coefficients.
FilterLength	10	Reports the length of the filter, the number of coefficients or taps.
Leakage	0	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1. Defaults to 0.

Property	Default Value	Description
PersistentMemory	false or true	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to false.
States	zeros(l-1,1)	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l - 1).
StepSize	0.1	Sets the SD algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

Examples   Adaptive line enhancement using a 32-coefficient FIR filter to perform the enhancement. This example runs for 5000 iterations, as you see in property iter.

```
d  = 1;                      % Number of samples of delay
ntr= 5000;                   % Number of iterations
v  = sin(2*pi*0.05*[1:ntr+d]); % Sinusoidal signal
n  = randn(1,ntr+d);         % Noise signal
```

```
x  = v(1:ntr)+n(1:ntr);          % Input signal
d  = v(1+d:ntr+d)+n(1+d:ntr+d);  % Desired signal
mu = 0.0001;                     % Sign-data step size.
ha = adaptfilt.sd(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:ntr,[d;y;v(1:end-1)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of a Noisy Sinusoidal Signal');
legend('Observed','Enhanced','Original');
 xlabel('Time Index'); ylabel('Signal Value');
[pxx,om] = pwelch(x(ntr-1000:ntr));
 pyy = pwelch(y(ntr-1000:ntr));
subplot(2,1,2);
plot(om/pi,10*log10([pxx/max(pxx),pyy/max(pyy)]));
axis([0 1 -60 20]); legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```

Each of the variants — sign-data, sign-error, and sign-sign — uses the same example. You can compare the results by viewing the figure shown for each adaptive filter method — adaptfilt.sd, adaptfilt.se, and adaptfilt.ss.

References   Moschner, J.L., "Adaptive Filter with Clipped Input Data," Ph.D. thesis, Stanford Univ., Stanford, CA, June 1970.

Hayes, M., *Statistical Digital Signal Processing and Modeling,* New York Wiley, 1996.

See Also   adaptfilt.lms | adaptfilt.se | adaptfilt.ss

adaptfilt.se

Purpose FIR adaptive filter that uses sign-error algorithm

Syntax `ha = adaptfilt.se(l,step,leakage,coeffs,states)`

Description `ha = adaptfilt.se(l,step,leakage,coeffs,states)` constructs an FIR sign-error adaptive filter `ha`.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.se`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	SE step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.1
`leakage`	Your SE leakage factor. It must be a scalar between 0 and 1. When `leakage` is less than one, `adaptfilt.se` implements a leaky SE algorithm. When you omit the `leakage` property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
`coeffs`	Vector of initial filter coefficients. it must be a length `l` vector. `coeffs` defaults to length `l` vector with elements equal to zero.
`states`	Vector of initial filter states for the adaptive filter. It must be a length `l-1` vector. `states` defaults to a length `l-1` vector of zeros.

Properties   In the syntax for creating the adaptfilt object, the input options are properties of the object you create. This table lists the properties for the sign-error SD object, their default values, and a brief description of the property.

Property	Default Value	Description
Algorithm	Sign-error	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	zeros(1,l)	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input. Should be initialized with the initial coefficients for the FIR filter prior to adapting.
FilterLength	10	Reports the length of the filter, the number of coefficients or taps
Leakage	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1. Defaults to one if omitted.

Property	Default Value	Description
PersistentMemory	false or true	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to false.
States	zeros(l-1,1)	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l -1).
StepSize	0.1	Sets the SE algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

Use inspect(ha) to view or change the object properties graphically using the MATLAB Property Inspector.

Examples   Adaptive line enhancement using a 32-coefficient FIR filter running over 5000 iterations.

```
d = 1;                      % Number of samples of delay
ntr= 5000;                  % Number of iterations
v = sin(2*pi*0.05*(1:ntr+d));    % Sinusoidal signal
n = randn(1,ntr+d);         % Noise signal
```

```
x = v(1:ntr)+n(1:ntr);        % Input signal --(delayed desired signal)
d = v(1+d:ntr+d)+n(1+d:ntr+d);  % Desired signal
mu = 0.0001;                    % Sign-error step size
ha = adaptfilt.se(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1);
plot(1:ntr,[d;y;v(1:end-1)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of Noisy Sinusoid');
legend('Observed','Enhanced','Original');
xlabel('Time Index'); ylabel('Signal Value');
HWelch = spectrum.welch;
InputPsd = psd(HWelch,x(ntr-1000:ntr));
OutputPsd = psd(HWelch,y(ntr-1000:ntr));
CompPsdEst = [InputPsd.Data/max(InputPsd.Data), OutputPsd.Data/max(OutputPsd.Data)];
subplot(2,1,2); plot(InputPsd.Frequencies/pi,10*log10(CompPsdEst));
axis([0 1 -60 20]); legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```

Compare the figure shown here to the ones for `adaptfilt.sd` and
`adaptfilt.ss` to see how the variants perform on the same example.

References   Gersho, A, "Adaptive Filtering With Binary Reinforcement," IEEE Trans. Information Theory, vol. IT-30, pp. 191-199, March 1984.

Hayes, M, *Statistical Digital Signal Processing and Modeling*, New York, Wiley, 1996.

See Also   adaptfilt.sd | adaptfilt.ss | adaptfilt.lms

Purpose   FIR adaptive filter that uses sign-sign algorithm

Syntax   ha = adaptfilt.ss(l,step,leakage,coeffs,states)

Description   ha = adaptfilt.ss(l,step,leakage,coeffs,states) constructs an
FIR sign-error adaptive filter ha.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.ss.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. l defaults to 10.
step	SS step size. It must be a nonnegative scalar. step defaults to 0.1.
leakage	Your SS leakage factor. It must be a scalar between 0 and 1. When leakage is less than one, adaptfilt.lms implements a leaky SS algorithm. When you omit the leakage property in the calling syntax, it defaults to 1 providing no leakage in the adapting algorithm.
coeffs	Vector of initial filter coefficients. it must be a length l vector. coeffs defaults to length l vector with elements equal to zero.
states	Vector of initial filter states for the adaptive filter. It must be a length l -1 vector. states defaults to a length l-1 vector of zeros.

adaptfilt.ss can be called for a block of data, when x and d are vectors,
or in "sample by sample mode" using a For-loop with the method filter:

```
for n = 1:length(x)
```

```
ha = adaptfilt.ss(25,0.9);
[y(n),e(n)] = filter(ha,(x(n),d(n),s));
end
```

Properties

In the syntax for creating the adaptfilt object, most of the input options are properties of the object you create. This table lists the properties for sign-sign objects, their default values, and a brief description of the property.

Property	Default Value	Description
Algorithm	Sign-sign	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	zeros(1,l)	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input. Should be initialized with the initial coefficients for the FIR filter prior to adapting.
FilterLength	10	Reports the length of the filter, the number of coefficients or taps

Property	Default Value	Description
Leakage	1	Specifies the leakage parameter. Allows you to implement a leaky algorithm. Including a leakage factor can improve the results of the algorithm by forcing the algorithm to continue to adapt even after it reaches a minimum value. Ranges between 0 and 1. 1 is the default value.
PersistentMemory	false or true	Determine whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to false.
States	zeros(l-1,1)	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l-1).
StepSize	0.1	Sets the SE algorithm step size used for each iteration of the adapting algorithm. Determines both how quickly and how closely the adaptive filter converges to the filter solution.

Examples   Demonstrating adaptive line enhancement using a 32-coefficient FIR
filter provides a good introduction to the sign-sign algorithm.

```
d = 1;                          % Number of samples of delay
ntr= 5000;                      % Number of iterations
v = sin(2*pi*0.05*(1:ntr+d));    % Sinusoidal signal
n = randn(1,ntr+d);          % Noise signal
x = v(1:ntr)+n(1:ntr);        % Input signal --(delayed desired signal)
d = v(1+d:ntr+d)+n(1+d:ntr+d);  % Desired signal
mu = 0.0001;                      % Sign-error step size
ha = adaptfilt.ss(32,mu);
[y,e] = filter(ha,x,d);
subplot(2,1,1);
plot(1:ntr,[d;y;v(1:end-1)]);
axis([ntr-100 ntr -3 3]);
title('Adaptive Line Enhancement of Noisy Sinusoid');
legend('Observed','Enhanced','Original');
xlabel('Time Index'); ylabel('Signal Value');
HWelch = spectrum.welch;
InputPsd = psd(HWelch,x(ntr-1000:ntr));
OutputPsd = psd(HWelch,y(ntr-1000:ntr));
CompPsdEst = [InputPsd.Data/max(InputPsd.Data), OutputPsd.Data/max(OutputPsd.Data)];
subplot(2,1,2); plot(InputPsd.Frequencies/pi,10*log10(CompPsdEst));
axis([0 1 -60 20]); legend('Observed','Enhanced');
xlabel('Normalized Frequency (\times \pi rad/sample)');
ylabel('Power Spectral Density'); grid on;
```

This example is the same as the ones used for the sign-data and
sign-error examples. Comparing the figures shown for each of the
others lets you assess the performance of each for the same task.

References   Lucky, R.W, "Techniques For Adaptive Equalization of Digital
Communication Systems," Bell Systems Technical Journal, vol. 45, pp.
255-286, Feb. 1966

Hayes, M., *Statistical Digital Signal Processing and Modeling*, New
York, Wiley, 1996.

See Also   adaptfilt.se | adaptfilt.sd | adaptfilt.lms

adaptfilt.swftf

Purpose   FIR adaptive filter that uses sliding window fast transversal least squares

Syntax   ha = adaptfilt.swftf(l,delta,blocklen,gamma,gstates,
                dstates,...coeffs,states)

Description ha = adaptfilt.swftf(l,delta,blocklen,gamma,gstates,
                dstates,...coeffs,states) constructs a sliding window fast
                transversal least squares adaptive filter ha.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.swftf.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. l defaults to 10.
delta	Soft-constrained initialization factor. This scalar should be positive and sufficiently large to maintain stability. delta defaults to 1.
blocklen	Block length of the sliding window. This must be an integer at least as large as the filter length l, which is the default value.
gamma	Conversion factor. gamma defaults to the matrix [1 -1] that specifies soft-constrained initialization.
gstates	States of the Kalman gain updates. gstates defaults to a zero vector of length (l + blocklen - 1).
dstates	Desired signal states of the adaptive filter. dstates defaults to a zero vector of length equal to (blocklen - 1). For a default object, dstates is (l-1).

Input Argument	Description
coeffs	Vector of initial filter coefficients. It must be a length l vector. coeffs defaults to length l vector of all zeros.
states	Vector of initial filter states. states defaults to a zero vector of length equal to (l + blocklen - 2).

Properties

Since your adaptfilt.swftf filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.swftf objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
BkwdPredictions		Returns the predicted samples generated during adaptation. Refer to [2] in the bibliography for details about linear prediction.
BlockLength		Block length of the sliding window. This must be an integer at least as large as the filter length l, which is the default value.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.

Name	Range	Description
ConversionFactor		Conversion factor. Called gamma when it is an input argument, it defaults to the matrix [1 -1] that specifies soft-constrained initialization.
DesiredSignal States		Desired signal states of the adaptive filter. dstates defaults to a zero vector with length equal to (blocklen - 1).
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
FwdPrediction		Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
InitFactor		Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. delta defaults to one.
KalmanGain		Empty when you construct the object, this gets populated after you run the filter.
KalmanGainStates		Contains the states of the Kalman gains for the adaptive algorithm. Initialized to a vector of double data type entries.

Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 + projectord - 2).

Examples

Over 500 iterations, perform a system identification of a 32-coefficient FIR filter.

```
x = randn(1,500);      % Input to the filter
b = fir1(31,0.5);      % FIR system to be identified
n = 0.1*randn(1,500);  % Observation noise signal
d = filter(b,1,x)+n;   % Desired signal
L = 32;                % Adaptive filter length
del = 0.1;             % Soft-constrained initialization factor
N = 64;                % block length
ha = adaptfilt.swftf(L,del,N);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
```

adaptfilt.swftf

```
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');  grid on;
```

Review the figure for the results of the example. When you evaluate the example you should get the same results, within the differences in the random noise signal you use.

References   Slock, D.T.M., and T. Kailath, "A Modular Prewindowing Framework for Covariance FTF RLS Algorithms," Signal Processing, vol. 28, pp. 47-61, 1992

Slock, D.T.M., and T. Kailath, "A Modular Multichannel Multi-Experiment Fast Transversal Filter RLS Algorithm," Signal Processing, vol. 28, pp. 25-45, 1992

See Also   adaptfilt.ftf | adaptfilt.swrls | adaptfilt.ap | adaptfilt.apru

Purpose   FIR adaptive filter that uses window recursive least squares (RLS)

Syntax   ha = adaptfilt.swrls(l,lambda,invcov,swblocklen,
                dstates,...coeffs,states)

Description ha = adaptfilt.swrls(l,lambda,invcov,swblocklen,
                dstates,...coeffs,states) constructs an FIR sliding window RLS
                adaptive filter ha.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.swrls.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps). It must be a positive integer. l defaults to 10.
lambda	RLS forgetting factor. This is a scalar and should lie within the range (0, 1]. lambda defaults to 1.
invcov	Inverse of the input signal covariance matrix. You should initialize invcov to a positive definite matrix.
swblocklen	Block length of the sliding window. This integer must be at least as large as the filter length. swblocklen defaults to 16.
dstates	Desired signal states of the adaptive filter. dstates defaults to a zero vector with length equal to (swblocklen - 1).
coeffs	Vector of initial filter coefficients. It must be a length l vector. coeffs defaults to length l vector of all zeros.
states	Vector of initial filter states. states defaults to a zero vector of length equal to (l + swblocklen - 2).

adaptfilt.swrls

Properties   Since your `adaptfilt.swrls` filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating `adaptfilt.swrls` objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
Coefficients	Any vector of l elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. `coeffs` defaults to length l vector of zeros when you do not provide the argument for input.
DesiredSignalStates	Vector	Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`swblocklen` - 1).
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps

Name	Range	Description
ForgettingFactor	Scalar	Forgetting factor of the adaptive filter. This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting forgetting factor = 1 denotes infinite memory while adapting to find the new filter. Note that this is the lambda input argument.
InvCov	Matrix	Square matrix with each dimension equal to the filter length l.
KalmanGain	Vector with dimensions (l,1)	Empty when you construct the object, this gets populated after you run the filter.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. Defaults to false.

adaptfilt.swrls

Name	Range	Description
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (1 + swblocklen - 2)
SwBlockLength	Integer	Block length of the sliding window. This integer must be at least as large as the filter length. swblocklen defaults to 16.

Examples   System Identification of a 32-coefficient FIR filter. Use 500 iterations to adapt to the unknown filter. After the example code, you see a figure that plots the results of the running the code.

```
x  = randn(1,500);      % Input to the filter
b  = fir1(31,0.5);      % FIR system to be identified
n  = 0.1*randn(1,500);  % Observation noise signal
d  = filter(b,1,x)+n;   % Desired signal
P0 = 10*eye(32);        % Initial correlation matrix inverse
lam = 0.99;             % RLS forgetting factor
N  = 64;                % Block length
ha = adaptfilt.swrls(32,lam,P0,N);
[y,e] = filter(ha,x,d);
subplot(2,1,1); plot(1:500,[d;y;e]);
 title('System Identification of an FIR Filter');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,1,2); stem([b.',ha.Coefficients.']);
legend('Actual','Estimated');
xlabel('Coefficient #'); ylabel('Coefficient Value');  grid on;
```

In the figure you see clearly that the adaptive filter process successfully identified the coefficients of the unknown FIR filter. You knew it

had to or many things that you take for granted, such as modems on computers, would not work.

See Also   adaptfilt.rls | adaptfilt.qrdrls | adaptfilt.hswrls

adaptfilt.tdafdct

Purpose   Adaptive filter that uses discrete cosine transform

Syntax   ha = adaptfilt.tdafdct(l,step,leakage,offset,delta,lambda,
                 coeffs,states)

Description ha = adaptfilt.tdafdct(l,step,leakage,offset,delta,lambda,
                 coeffs,states) constructs a transform-domain adaptive filter ha object
                 that uses the discrete cosine transform to perform filter adaptation.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.tdafdct.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. l defaults to 10.
step	Adaptive filter step size. It must be a nonnegative scalar. You can use maxstep to determine a reasonable range of step size values for the signals being processed. step defaults to 0.
leakage	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDCT algorithm. leakage defaults to 1 — no leakage.
offset	Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zero or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.

Input Argument	Description
delta	Initial common value of all of the transform domain powers. Its initial value should be positive. delta defaults to 5.
lambda	Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. lambda should lie between zero and one. For default filter objects, lambda equals (1 - step).
coeffs	Initial time domain coefficients of the adaptive filter. Set it to be a length l vector. coeffs defaults to a zero vector of length l.
states	Initial conditions of the adaptive filter. states defaults to a zero vector with length equal to (l - 1).

Properties   Since your adaptfilt.tdafdct filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.tdafdct objects. To show you the properties that apply, this table lists and describes each property for the transform domain filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation.
AvgFactor		Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. AvgFactor

Name	Range	Description
		should lie between zero and one. For default filter objects, `AvgFactor` equals (1 - `step`). `lambda` is the input argument that represent `AvgFactor`.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. `coeffs` defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps.
Leakage	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDFT algorithm. `leakage` defaults to 1 — no leakage.
Offset		Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zeros or by very small numbers when any of the FFT input signal powers become very small. `offset` defaults to zero.

Name	Range	Description
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
Power	2*l element vector	A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).
StepSize	0 to 1	Step size. It must be a nonnegative scalar, greater than zero and less than or equal to 1. You can use maxstep to determine a reasonable range of step size values for the signals being processed. step defaults to 0.

For checking the values of properties for an adaptive filter object, use get(ha) or enter the object name, without a trailing semicolon, at the MATLAB prompt.

Examples   Using 1000 iterations, perform a Quadrature Phase Shift Keying
(QPSK) adaptive equalization using a 32-coefficient FIR filter.

```
D  = 16;                         % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];       % Numerator coefficients of channel
a  = [1 -0.7];                   % Denominator coefficients of channel
ntr= 1000;                       % Number of iterations
s  = sign(randn(1,ntr+D)) + 1j*sign(randn(1,ntr+D)); %QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D)); % Noise signal
r  = filter(b,a,s)+n;            % Received signal
x  = r(1+D:ntr+D);               % Input signal (received signal)
d  = s(1:ntr);                   % Desired signal (delayed QPSK signal)
L  = 32;                         % filter length
mu = 0.01;                       % Step size
ha = adaptfilt.tdafdct(L,mu);
[y,e] = filter(ha,x,d);
subplot(2,2,1);
plot(1:ntr,real([d;y;e])); title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');
axis([-3 3 -3 3]); title('Received Signal Scatter Plot');
axis('square');  xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); grid on;
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]');
```

Compare the plots shown in this figure to those in the other time
domain filter variations. The comparison should help you select and
understand how the variants differ.

References Haykin, S.,*Adaptive Filter Theory*, 3rd Edition, Prentice Hall, N.J., 1996.

See Also adaptfilt.tdafdft | adaptfilt.fdaf | adaptfilt.blms

adaptfilt.tdafdft

Purpose   Adaptive filter that uses discrete Fourier transform

Syntax   ha = adaptfilt.tdafdft(l,step,leakage,offset,
                 delta,lambda,...coeffs,states)

Description ha = adaptfilt.tdafdft(l,step,leakage,offset,
                 delta,lambda,...coeffs,states) constructs a transform-domain
                 adaptive filter object ha using a discrete Fourier transform.

Input Arguments

Entries in the following table describe the input arguments for
adaptfilt.tdafdft.

Input Argument	Description
l	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. l defaults to 10.
step	Adaptive filter step size. It must be a nonnegative scalar. You can use maxstep to determine a reasonable range of step size values for the signals being processed. step defaults to 0.
leakage	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDFT algorithm. leakage defaults to 1 — no leakage.
offset	Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zeros or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.

Input Argument	Description
delta	Initial common value of all of the transform domain powers. Its initial value should be positive. delta defaults to 5.
lambda	Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. lambda should lie between zero and one. For default filter objects, LAMBDA equals (1 - step).
coeffs	Initial time domain coefficients of the adaptive filter. Set it to be a length l vector. coeffs defaults to a zero vector of length l.
states	Initial conditions of the adaptive filter. states defaults to a zero vector with length equal to (l - 1).

Properties   Since your adaptfilt.tdafdft filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.tdafdft objects. To show you the properties that apply, this table lists and describes each property for the transform domain filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
AvgFactor		Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. AvgFactor

adaptfilt.tdafdft

Name	Range	Description
		should lie between zero and one. For default filter objects, AvgFactor equals (1 - step). lambda is the input argument that represent AvgFactor.
Coefficients	Vector of elements	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps
Leakage	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the TDAFDFT algorithm. leakage defaults to 1 — no leakage.
Offset		Offset for the normalization terms in the coefficient updates. You can use this argument to avoid dividing by zeros or by very small numbers when any of the FFT input signal powers become very small. offset defaults to zero.

Name	Range	Description
PersistentMemory	false or true	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.
Power	2*l element vector	A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
States	Vector of elements, data type double	Vector of the adaptive filter states. states defaults to a vector of zeros which has length equal to (l + projectord - 2).
StepSize	0 to 1	Step size. It must be a nonnegative scalar, greater than zero and less than or equal to 1. step defaults to 0.

Examples   Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient FIR filter (1000 iterations).

```
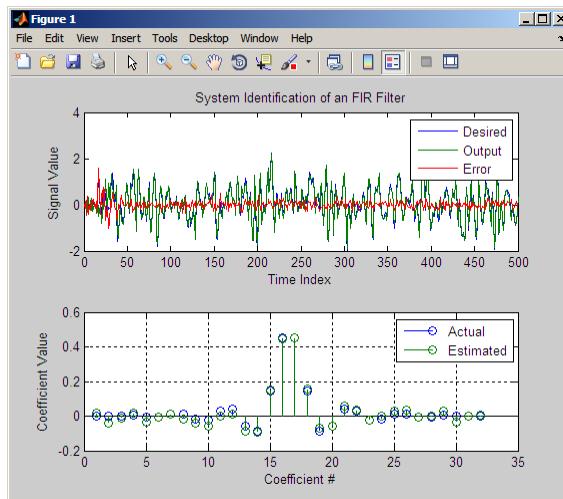D = 16;                        % Number of samples of delay
b = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a = [1 -0.7];                  % Denominator coefficients of channel
ntr= 1000;                     % Number of iterations
```

```
s  = sign(randn(1,ntr+D)) + 1j*sign(randn(1,ntr+D));% Baseband QPSK signal
n  = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));      % Noise signal
r  = filter(b,a,s)+n;           % Received signal
x  = r(1+D:ntr+D);              % Input signal (received signal)
d  = s(1:ntr);                  % Desired signal (delayed QPSK signal)
L  = 32;                        % filter length
mu = 0.01;                      % Step size
ha = adaptfilt.tdafdft(L,mu);
[y,e] = filter(ha,x,d);
subplot(2,2,1); plot(1:ntr,real([d;y;e]));
title('In-Phase Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components');
legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.');
 axis([-3 3 -3 3]); title('Received Signal Scatter Plot');
axis('square');  xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.');
axis([-3 3 -3 3]); title('Equalized Signal Scatter Plot');
axis('square'); xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```

All of the time domain adaptive filter reference pages use this QPSK example. By comparing the results for each variation you get an idea of the differences in the way each one performs.

This figure demonstrates the results of running the example code shown.

References   Haykin, S.,*Adaptive Filter Theory*, 3rd Edition, Prentice Hall, N.J., 1996

See Also   `adaptfilt.tdafdct` | `adaptfilt.fdaf` | `adaptfilt.blms`

adaptfilt.ufdaf

Purpose	FIR adaptive filter that uses unconstrained frequency-domain with quantized step size normalization
Syntax	`ha = adaptfilt.ufdaf(l,step,leakage,delta,lambda,blocklen, offset,coeffs,states)`
Description	`ha = adaptfilt.ufdaf(l,step,leakage,delta,lambda,blocklen, offset,coeffs,states)`

constructs an unconstrained frequency-domain FIR adaptive filter `ha` with quantized step size normalization.

Input Arguments

Entries in the following table describe the input arguments for `adaptfilt.ufdaf`.

Input Argument	Description
`l`	Adaptive filter length (the number of coefficients or taps) and it must be a positive integer. `l` defaults to 10.
`step`	Adaptive filter step size. It must be a nonnegative scalar. `step` defaults to 0.
`leakage`	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the UFDAF algorithm. `leakage` defaults to 1 — no leakage.
`delta`	Initial common value of all of the FFT input signal powers. the initial value of delta should be positive, and it defaults to 1.

Input Argument	Description
lambda	Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. lambda should lie in the range (0,1]. For default UFDAF filter objects, lambda defaults to 0.9.
blocklen	Block length for the coefficient updates. This must be a positive integer. For faster execution, (blocklen l) should be a power of two. blocklen defaults to l.
offset	Offset for the normalization terms in the coefficient updates. This can help you avoid divide by zero conditions, or divide by very small numbers conditions, when any of the FFT input signal powers become very small. Default value is zero.
coeffs	Initial time-domain coefficients of the adaptive filter. It should be a length l vector. The filter object uses these coefficients to compute the initial frequency-domain filter coefficients via an FFT computed after zero-padding the time-domain vector by blocklen.
states	Adaptive filter states. states defaults to a zero vector with length equal to l.

Properties   Since your adaptfilt.ufdaf filter is an object, it has properties that define its behavior in operation. Note that many of the properties are also input arguments for creating adaptfilt.ufdaf objects. To show you the properties that apply, this table lists and describes each property for the filter object.

Name	Range	Description
Algorithm	None	Defines the adaptive filter algorithm the object uses during adaptation
AvgFactor		Specifies the averaging factor used to compute the exponentially-windowed FFT input signal powers for the coefficient updates. AvgFactor should lie in the range (0,1]. For default UFDAF filter objects, AvgFactor defaults to 0.9. Note that AvgFactor and lambda are the same thing — lambda is an input argument and AvgFactor a property of the object.
BlockLength		Block length for the coefficient updates. This must be a positive integer. For faster execution, (blocklen + 1) should be a power of two. blocklen defaults to 1.
FFTCoefficients		Stores the discrete Fourier transform of the filter coefficients in coeffs.
FFTStates		States for the FFT operation.
FilterLength	Any positive integer	Reports the length of the filter, the number of coefficients or taps

Name	Range	Description
Leakage	0 to 1	Leakage parameter of the adaptive filter. When you set this argument to a value between zero and one, you are implementing a leaky version of the UFDAF algorithm. leakage defaults to 1 — no leakage.
Offset		Offset for the normalization terms in the coefficient updates. This can help you avoid divide by zero conditions, or divide by very small numbers conditions, when any of the FFT input signal powers become very small. Default value is zero.
PersistentMemory	false or true	Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to false.

Name	Range	Description
Power	2*1 element vector	A vector of 2*1 elements, each initialized with the value `delta` from the input arguments. As you filter data, `Power` gets updated by the filter process.
StepSize	0 to 1	Adaptive filter step size. It must be a nonnegative scalar. You can use `maxstep` to determine a reasonable range of step size values for the signals being processed. `step` defaults to 0.

Examples

Show an example of Quadrature Phase Shift Keying (QPSK) adaptive equalization using a 32-coefficient adaptive filter. For fidelity, use 1024 iterations. The figure that follows the code provides the information you need to assess the performance of the equalization process.

```
D  = 16;                        % Number of samples of delay
b  = exp(1j*pi/4)*[-0.7 1];      % Numerator coefficients of channel
a  = [1 -0.7];                   % Denominator coefficients of channel
ntr= 1024;                       % Number of iterations
s = sign(randn(1,ntr+D))+1j*sign(randn(1,ntr+D));% Baseband QPSK signal
n = 0.1*(randn(1,ntr+D) + 1j*randn(1,ntr+D));
r = filter(b,a,s)+n;% Received signal
x = r(1+D:ntr+D);   % Input signal (received signal)
d = s(1:ntr);        % Desired signal (delayed QPSK signal)
del = 1;            %Initial FFT input powers
mu  = 0.1;           % Step size
lam = 0.9;           % Averaging factor
ha = adaptfilt.ufdaf(32,mu,1,del,lam);
[y,e] = filter(ha,x,d);
subplot(2,2,1);  plot(1:1000,real([d(1:1000);y(1:1000);e(1:1000)])));
```

```
title('In-Phase Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,2); plot(1:ntr,imag([d;y;e]));
title('Quadrature Components'); legend('Desired','Output','Error');
xlabel('Time Index'); ylabel('Signal Value');
subplot(2,2,3); plot(x(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Received Signal Scatter Plot'); axis('square');
xlabel('Real[x]'); ylabel('Imag[x]'); grid on;
subplot(2,2,4); plot(y(ntr-100:ntr),'.'); axis([-3 3 -3 3]);
title('Equalized Signal Scatter Plot'); axis('square');
xlabel('Real[y]'); ylabel('Imag[y]'); grid on;
```


References   Shynk, J.J.,"Frequency-domain and Multirate Adaptive Filtering,"
IEEE Signal Processing Magazine, vol. 9, no. 1, pp. 14-37, Jan. 1992

See Also   adaptfilt.fdaf | adaptfilt.pbufdaf | adaptfilt.blms |
adaptfilt.blmsfft

allpassbpc2bpc

Purpose   Allpass filter for complex bandpass transformation

Syntax   [AllpassNum,AllpassDen] = allpassbpc2bpc(Wo,Wt)

Description   [AllpassNum,AllpassDen] = allpassbpc2bpc(Wo,Wt) returns the
numerator, AllpassNum, and the denominator, AllpassDen, of the
first-order allpass mapping filter for performing a complex bandpass
to complex bandpass frequency transformation. This transformation
effectively places two features of an original filter, located at frequencies
W_{o1} and W_{o2}, at the required target frequency locations W_{t1} and W_{t2}.
It is assumed that W_{t2} is greater than W_{t1}. In most of the cases the
features selected for the transformation are the band edges of the
filter passbands. In general it is possible to select any feature; e.g.,
the stopband edge, the DC, the deep minimum in the stopband, or
other ones.

Relative positions of other features of an original filter do not change in
the target filter. This means that it is possible to select two features of
an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
precede F_2 after the transformation. However, the distance between F_1
and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of
filters; e.g., complex notch filters or resonators can be repositioned at
two distinct desired frequencies at any place around the unit circle.
This is very attractive for adaptive systems.

Examples   Design the allpass mapping filter changing the complex bandpass filter
with the band edges at W_{o1}=0.2 and W_{o2}=0.4 to the new band edges
of W_{t1}=0.3 and W_{t2}=0.6. Find the frequency response of the allpass
mapping filter:

```
Wo = [0.2, 0.4]; Wt = [0.3, 0.6];
[AllpassNum, AllpassDen] = allpassbpc2bpc(Wo, Wt);
[ha, f] = freqz(AllpassNum, AllpassDen,'whole');
plot(f/pi,angle(ha)/pi, Wt, Wo, 'ro'); title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```

Arguments

Variable	Description
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also   iirbpc2bpc | zpkbpc2bpc

allpasslp2bp

Purpose   Allpass filter for lowpass to bandpass transformation

Syntax   `[AllpassNum,AllpassDen] = allpasslp2bp(Wo,Wt)`

Description   `[AllpassNum,AllpassDen] = allpasslp2bp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the second-order allpass mapping filter for performing a real lowpass to real bandpass frequency transformation. This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "DC mobility," which means that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of W_t.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and repositioned at two distinct desired frequencies.

Examples   Design the allpass mapping filter changing the lowpass filter with cutoff frequency at W_o=0.5 to the real–valued bandpass filter with cutoff frequencies at W_{t1}0.25 and W_{t2}=0.375.

Compute the frequency response and plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
Wo = 0.5; Wt = [0.25, 0.375];
[AllpassNum, AllpassDen] = allpasslp2bp(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```


Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter

allpasslp2bp

Variable	Description
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also   iirlp2bp | zpklp2bp

Purpose   Allpass filter for lowpass to complex bandpass transformation

Syntax   [AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt)

Description [AllpassNum,AllpassDen] = allpasslp2bpc(Wo,Wt) returns the
numerator, AllpassNum, and the denominator, AllpassDen, of the
first-order allpass mapping filter for performing a real lowpass to
complex bandpass frequency transformation. This transformation
effectively places one feature of an original filter, located at frequency
-W_o, at the required target frequency location, W_{t1}, and the second
feature, originally at +W_o, at the new location, W_{t2}. It is assumed that
W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in
the target filter. This means that it is possible to select two features of
an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
precede F_2 after the transformation. However, the distance between F_1
and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation
is not restricted only to the cutoff frequency of an original lowpass filter.
In general it is possible to select any feature; e.g., the stopband edge,
the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming
other types of filters; e.g., real notch filters or resonators can be doubled
and positioned at two distinct desired frequencies at any place around
the unit circle forming a pair of complex notches/resonators. This
transformation can be used for designing bandpass filters for radio
receivers from the high-quality prototype lowpass filter.

Examples   Design the allpass mapping filter changing the real lowpass filter with
the cutoff frequency of W_o=0.5 into a complex bandpass filter with band
edges of W_{t1}=0.2 and W_{t2}=0.4 precisely defined. Calculate the frequency
response of the mapping filter in the full range:

```
Wo = 0.5; Wt = [0.2,0.4];
[AllpassNum, AllpassDen] = allpasslp2bpc(Wo, Wt);
```

allpasslp2bpc

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also   iirlp2bpc | zpklp2bpc

Purpose   Allpass filter for lowpass to bandstop transformation

Syntax   [AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt)

Description [AllpassNum,AllpassDen] = allpasslp2bs(Wo,Wt) returns the
numerator, AllpassNum, and the denominator, AllpassDen, of the
second-order allpass mapping filter for performing a real lowpass to real
bandstop frequency transformation. This transformation effectively
places one feature of an original filter, located at frequency $-W_o$, at
the required target frequency location, W_{t1}, and the second feature,
originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater
than W_{t1}. This transformation implements the "Nyquist Mobility," which
means that the DC feature stays at DC, but the Nyquist feature moves
to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter change in the
target filter. This means that it is possible to select two features
of an original filter, F_1 and F_2, with F_1 preceding F_2. After the
transformation feature F_2 will precede F_1 in the target filter. However,
the distance between F_1 and F_2 will not be the same before and after
the transformation.

Choice of the feature subject to the lowpass to bandstop transformation
is not restricted only to the cutoff frequency of an original lowpass filter.
In general it is possible to select any feature; e.g., the stopband edge,
the DC, the deep minimum in the stopband, or other ones.

Examples   Design the allpass filter changing the lowpass filter with cutoff
frequency at W_o=0.5 to the real bandstop filter with cutoff frequencies
at W_{t1}=0.25 and W_{t2}=0.375:

```
Wo = 0.5; Wt = [0.25, 0.375];
[AllpassNum, AllpassDen] = allpasslp2bs(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```

In the figure, you find the mapping filter function as determined by the example. Note the response is normalized to π:

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also   iirlp2bs | zpklp2bs

allpasslp2bsc

Purpose   Allpass filter for lowpass to complex bandstop transformation

Syntax   `[AllpassNum,AllpassDen] = allpasslp2bsc(Wo,Wt)`

Description   `[AllpassNum,AllpassDen] = allpasslp2bsc(Wo,Wt)` returns the
                  numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the
                  first-order allpass mapping filter for performing a real lowpass to
                  complex bandstop frequency transformation. This transformation
                  effectively places one feature of an original filter, located at frequency
                  $-W_o$, at the required target frequency location, W_{t1}, and the second
                  feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed
                  that W_{t2} is greater than W_{t1}. Additionally the transformation swaps
                  passbands with stopbands in the target filter.

                  Relative positions of other features of an original filter do not change in
                  the target filter. This means that it is possible to select two features of
                  an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
                  precede F_2 after the transformation. However, the distance between F_1
                  and F_2 will not be the same before and after the transformation.

                  Choice of the feature subject to the lowpass to bandstop transformation
                  is not restricted only to the cutoff frequency of an original lowpass filter.
                  In general it is possible to select any feature; e.g., the stopband edge,
                  the DC, the deep minimum in the stopband, or other ones.

                  Lowpass to bandpass transformation can also be used for transforming
                  other types of filters; e.g., real notch filters or resonators can be doubled
                  and positioned at two distinct desired frequencies at any place around
                  the unit circle forming a pair of complex notches/resonators. This
                  transformation can be used for designing bandstop filters for band
                  attenuation or frequency equalizers, from the high-quality prototype
                  lowpass filter.

Examples   Design the allpass filter changing the real lowpass filter with the cutoff
                  frequency of W_o=`0.5` into a complex bandstop filter with band edges of
                  W_{t1}=`0.2` and W_{t2}=`0.4` precisely defined:

                  `Wo = 0.5; Wt = [0.2,0.4];`

```
[AllpassNum, AllpassDen] = allpasslp2bsc(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also   iirlp2bsc | zpklp2bsc

allpasslp2hp

Purpose	Allpass filter for lowpass to highpass transformation
Syntax	`[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)`

Description   `[AllpassNum,AllpassDen] = allpasslp2hp(Wo,Wt)` returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the first-order allpass mapping filter for performing a real lowpass to real highpass frequency transformation. This transformation effectively places one feature of an original filter, located originally at frequency, W_o, at the required target frequency location, W_t, at the same time rotating the whole frequency response by half of the sampling frequency. Result is that the DC and Nyquist features swap places.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to highpass transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband.

Lowpass to highpass transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can change their position in a simple way by using the lowpass to highpass transformation.

Examples   Design the allpass filter changing the lowpass filter to the highpass filter with its cutoff frequency moved from W_o=0.5 to W_t=0.25.

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2hp(Wo, Wt);
```

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt');
ylabel('Old Frequency, Wo');
```

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

See Also   iirlp2hp | zpklp2hp

allpasslp2lp

Purpose   Allpass filter for lowpass to lowpass transformation

Syntax   `[AllpassNum,AllpassDen] = allpasslp2lp(Wo,Wt)`

Description   `[AllpassNum,AllpassDen] = allpasslp2lp(Wo,Wt)` returns the
numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the
first-order allpass mapping filter for performing a real lowpass to real
lowpass frequency transformation. This transformation effectively
places one feature of an original filter, located originally at frequency
W_o, at the required target frequency location, W_t.

Relative positions of other features of an original filter do not change in
the target filter. This means that it is possible to select two features of
an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
precede F_2 after the transformation. However, the distance between F_1
and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to lowpass transformation
is not restricted to the cutoff frequency of an original lowpass filter. In
general it is possible to select any feature; e.g., the stopband edge, the
DC, the deep minimum in the stopband and so on.

Lowpass to lowpass transformation can also be used for transforming
other types of filters; e.g., notch filters or resonators can change
their position in a simple way by applying the lowpass to lowpass
transformation.

Examples   Design the allpass filter changing the lowpass filter cutoff frequency
originally at W_o=0.5 to W_t=0.25. Plot the phase response normalized to
π, which is in effect the mapping function $W_o(W_t)$. Please note that the
transformation works in the same way for both positive and negative
frequencies:

```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpasslp2lp(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
```

```
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```


As shown in the figure, `allpasslp2lp` generates a mapping function that converts your prototype lowpass filter to a target lowpass filter with different passband specifications.

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter

allpasslp2lp

Variable	Description
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References
Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

See Also
iirlp2lp | zpklp2lp

Purpose　　Allpass filter for lowpass to M-band transformation

Syntax　　　[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt)
[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt,Pass)

Description　[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt) returns the numerator, AllpassNum, and the denominator, AllpassDen, of the Mth-order allpass mapping filter for performing a real lowpass to real multipassband frequency transformation. Parameter M is the number of times an original feature is replicated in the target filter. This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, $W_{t1},...,W_{tM}$. By default the DC feature is kept at its original location.

[AllpassNum,AllpassDen] = allpasslp2mb(Wo,Wt,Pass) allows you to specify an additional parameter, Pass, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without redesigning them. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

allpasslp2mb

Examples   Design the allpass filter changing the real lowpass filter with the cutoff frequency of W_o=0.5 into a real multiband filter with band edges of W_t=[1:2:9]/10 precisely defined. Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
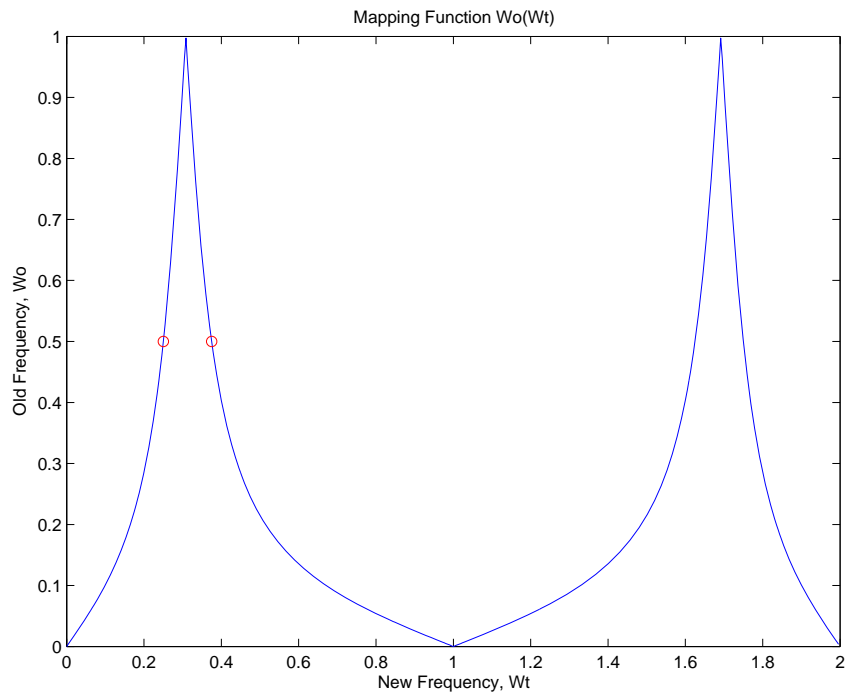Wo = 0.5; Wt = [1:2:9]/10;
[AllpassNum, AllpassDen] = allpasslp2mb(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice ('pass'/'stop') of passband/stopband at DC, 'pass' being the default
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems,"*MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

Feyh, G., J.C. Franchitti and C.T. Mullis, "All-pass filter interpolation and frequency transformation problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachusetts, Addison-Wesley, 1987.

Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations, Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

See Also   iirlp2mb | zpklp2mb

allpasslp2mbc

Purpose   Allpass filter for lowpass to complex M-band transformation

Syntax   [AllpassNum,AllpassDen] = allpasslp2mbc(Wo,Wt)

Description   [AllpassNum,AllpassDen] = allpasslp2mbc(Wo,Wt) returns the
                   numerator, AllpassNum, and the denominator, AllpassDen, of the
                   Mth-order allpass mapping filter for performing a real lowpass to
                   complex multipassband frequency transformation. Parameter M is the
                   number of times an original feature is replicated in the target filter.
                   This transformation effectively places one feature of an original filter,
                   located at frequency W_o, at the required target frequency locations,
                   $W_{t1},...,W_{tM}$.

                   Relative positions of other features of an original filter do not change in
                   the target filter. This means that it is possible to select two features of
                   an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
                   precede F_2 after the transformation. However, the distance between F_1
                   and F_2 will not be the same before and after the transformation.

                   Choice of the feature subject to this transformation is not restricted
                   to the cutoff frequency of an original lowpass filter. In general it is
                   possible to select any feature; e.g., the stopband edge, the DC, the deep
                   minimum in the stopband, or other ones.

                   This transformation can also be used for transforming other types of
                   filters; e.g., notch filters or resonators can be easily replicated at a
                   number of required frequency locations without the need to design them
                   again. A good application would be an adaptive tone cancellation circuit
                   reacting to the changing number and location of tones.

Examples   Design the allpass filter changing the real lowpass filter with the cutoff
                   frequency of Wo=0.5 into a complex multiband filter with band edges of
                   Wt=[-3+1:2:9]/10 precisely defined:

                   ```
                   Wo = 0.5; Wt = [-3+1:2:9]/10;
                   [AllpassNum, AllpassDen] = allpasslp2mbc(Wo, Wt);
                   ```

                   Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, angle(h)/pi, Wt, Wo, 'ro');
 title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```

In this example, the resulting mapping function converts real filters to multiband complex filters.

allpasslp2mbc

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also   iirlp2mbc | zpklp2mbc

Purpose   Allpass filter for lowpass to complex N-point transformation

Syntax   `[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)`

Description `[AllpassNum,AllpassDen] = allpasslp2xc(Wo,Wt)` returns the
numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the
`Nth`-order allpass mapping filter, where `N` is the allpass filter order,
for performing a real lowpass to complex multipoint frequency
transformation. Parameter `N` also specifies the number of replicas of the
prototype filter created around the unit circle after the transformation.
This transformation effectively places `N` features of the, original filter
located at frequencies $W_{o1},...,W_{oN}$, at the required target frequency
locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same
in the target filter for the Nyquist mobility and are reversed for the
DC mobility. For the Nyquist mobility this means that it is possible to
select two features of an original filter, F_1 and F_2, with F_1 preceding F_2.
Feature F_1 will still precede F_2 after the transformation. However,
the distance between F_1 and F_2 will not be the same before and after
the transformation. For DC mobility feature F_2 will precede F_1 after
the transformation.

Choice of the feature subject to this transformation is not restricted
to the cutoff frequency of an original lowpass filter. In general it is
possible to select any feature; e.g., the stopband edge, the DC, the deep
minimum in the stopband, or other ones. The only condition is that the
features must be selected in such a way that when creating `N` bands
around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of
filters; e.g., notch filters or resonators can be easily replicated at a
number of required frequency locations. A good application would be
an adaptive tone cancellation circuit reacting to the changing number
and location of tones.

Examples   Design the allpass filter moving four features of an original complex
filter given in `W`o to the new independent frequency locations `W`t. Please

note that the transformation creates N replicas of an original filter around the unit circle, where N is the order of the allpass mapping filter:

```
Wo = [-0.2, 0.3, -0.7, 0.4]; Wt = [0.3, 0.5, 0.7, 0.9];
[AllpassNum, AllpassDen] = allpasslp2xc(Wo, Wt);
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Arguments

Variable	Description
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also   iirlp2xc | zpklp2xc

Purpose   Allpass filter for lowpass to N-point transformation

Syntax   [AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt)
                  [AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt,Pass)

Description   [AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt) returns the
                  numerator, AllpassNum, and the denominator, AllpassDen, of the
                  Nth-order allpass mapping filter, where N is the allpass filter order, for
                  performing a real lowpass to real multipoint frequency transformation.
                  Parameter N also specifies the number of replicas of the prototype
                  filter created around the unit circle after the transformation. This
                  transformation effectively places N features of an original filter, located
                  at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations,
                  $W_{t1},...,W_{tM}$. By default the DC feature is kept at its original location.

                  [AllpassNum,AllpassDen] = allpasslp2xn(Wo,Wt,Pass) allows
                  you to specify an additional parameter, Pass, which chooses between
                  using the "DC Mobility" and the "Nyquist Mobility." In the first case
                  the Nyquist feature stays at its original location and the DC feature is
                  free to move. In the second case the DC feature is kept at an original
                  frequency and the Nyquist feature is movable.

                  Relative positions of other features of an original filter are the same
                  in the target filter for the Nyquist mobility and are reversed for the
                  DC mobility. For the Nyquist mobility this means that it is possible to
                  select two features of an original filter, F_1 and F_2, with F_1 preceding F_2.
                  Feature F_1 will still precede F_2 after the transformation. However,
                  the distance between F_1 and F_2 will not be the same before and after
                  the transformation. For DC mobility feature F_2 will precede F_1 after
                  the transformation.

                  Choice of the feature subject to this transformation is not restricted
                  to the cutoff frequency of an original lowpass filter. In general it is
                  possible to select any feature; e.g., the stopband edge, the DC, the deep
                  minimum in the stopband, or other ones. The only condition is that the
                  features must be selected in such a way that when creating N bands
                  around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations without the need of designing them again. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Arguments

Variable	Description
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice ('pass'/'stop') of passband/stopband at DC, 'pass' being the default
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Cain, G.D., A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

Krukowski, A., G.D. Cain and I. Kale, "Custom designed high-order frequency transformations for IIR filters," *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

See Also   iirlp2xn | zpklp2xn

allpassrateup

Purpose	Allpass filter for integer upsample transformation

Syntax   `[AllpassNum,AllpassDen] = allpassrateup(N)`

Description   `[AllpassNum,AllpassDen] = allpassrateup(N)` returns the
numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the
Nth-order allpass mapping filter for performing the rateup frequency
transformation, which creates `N` equal replicas of the prototype filter
frequency response.

Relative positions of other features of an original filter do not change in
the target filter. This means that it is possible to select two features of
an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
precede F_2 after the transformation. However, the distance between F_1
and F_2 will not be the same before and after the transformation.

Examples   Design the allpass filter creating the effect of upsampling the digital
filter four times:

Choose any feature from an original filter, say at W_o=`0.2`:

```
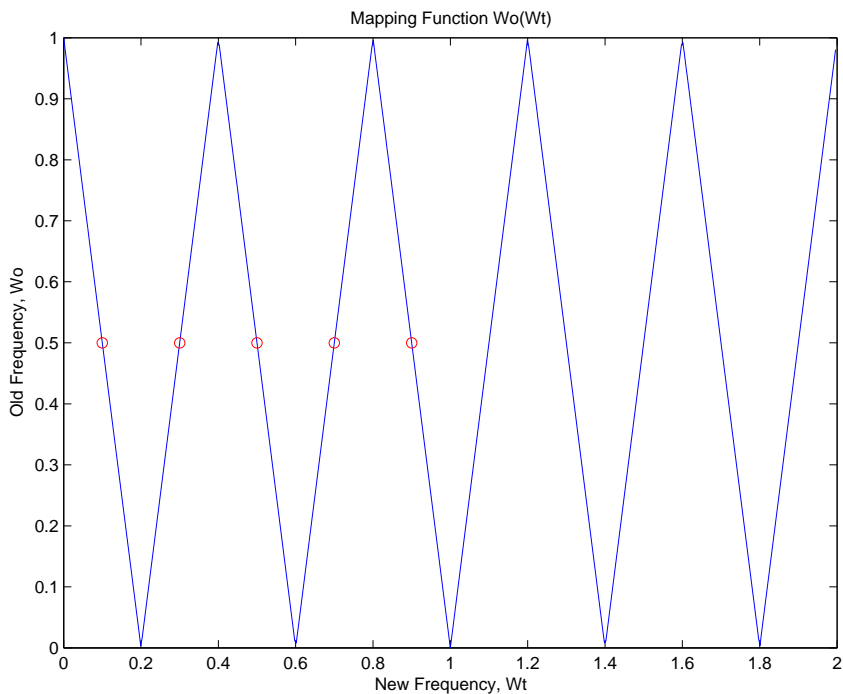N = 4;
Wo = 0.2; Wt = Wo/N + 2*[0:N-1]/N;
[AllpassNum, AllpassDen] = allpassrateup(N);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Arguments

Variable	Description
N	Frequency replication ratio (upsampling ratio)
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also   `iirrateup` | `zpkrateup`

7-187

allpassshift

Purpose   Allpass filter for real shift transformation

Syntax   `[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)`

Description   `[AllpassNum,AllpassDen] = allpassshift(Wo,Wt)` returns the
numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the
second-order allpass mapping filter for performing a real frequency shift
transformation. This transformation places one selected feature of an
original filter, located at frequency W_o, at the required target frequency
location, W_t. This transformation implements the "DC mobility," which
means that the Nyquist feature stays at Nyquist, but the DC feature
moves to a location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in
the target filter. This means that it is possible to select two features of
an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
precede F_2 after the transformation. However, the distance between F_1
and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not
restricted to the cutoff frequency of an original lowpass filter. In general
it is possible to select any feature; e.g., the stopband edge, the DC, the
deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types
of filters; e.g., notch filters or resonators can be moved to a different
frequency by applying a shift transformation. In such a way you can
avoid designing the filter from the beginning.

Examples   Design the allpass filter precisely shifting one feature of the lowpass
filter originally at W_o=`0.5` to the new frequencies of W_t=`0.25`:

```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpassshift(Wo, Wt);
```

Calculate the frequency response of the mapping filter in the full range:

```
[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Plot the phase response normalized to π, which is in effect the mapping function $W_o(W_t)$. Please note that the transformation works in the same way for both positive and negative frequencies:

```
plot(f/pi, abs(angle(h))/pi, Wt, Wo, 'ro');
title('Mapping Function Wo(Wt)');
xlabel('New Frequency, Wt'); ylabel('Old Frequency, Wo');
```

allpassshift

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also   iirshift | zpkshift

Purpose   Allpass filter for complex shift transformation

Syntax   [AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt)
                   [AllpassNum,AllpassDen] = allpassshiftc(0,0.5)
                   [AllpassNum,AllpassDen] = allpassshiftc(0,-0.5)

Description   [AllpassNum,AllpassDen] = allpassshiftc(Wo,Wt) returns the
                   numerator, AllpassNum, and denominator, AllpassDen, vectors of the
                   allpass mapping filter for performing a complex frequency shift of the
                   frequency response of the digital filter by an arbitrary amount.

                   [AllpassNum,AllpassDen] = allpassshiftc(0,0.5) calculates
                   the allpass filter for doing the Hilbert transformation, a 90 degree
                   counterclockwise rotation of an original filter in the frequency domain.

                   [AllpassNum,AllpassDen] = allpassshiftc(0,-0.5) calculates the
                   allpass filter for doing an inverse Hilbert transformation, i.e. a 90
                   degree clockwise rotation of an original filter in the frequency domain.

Examples   Design the allpass filter precisely rotating the whole filter by the
                  amount defined by the location of the selected feature from an original
                  filter, W_o=0.5, and its required position in the target filter, W_t=0.25:

```
Wo = 0.5; Wt = 0.25;
[AllpassNum, AllpassDen] = allpassshiftc(Wo, Wt);
```

                  Calculate the frequency response of the mapping filter in the full range:

```
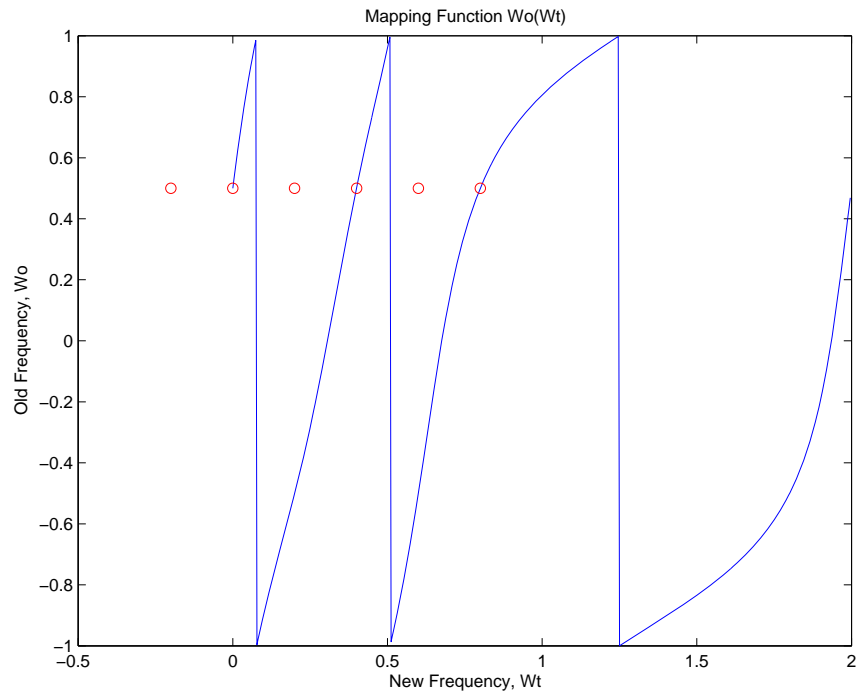[h, f] = freqz(AllpassNum, AllpassDen, 'whole');
```

Arguments

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter

allpassshiftc

Variable	Description
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

References   Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On Digital Differentiators, Hilbert Transformers, and Half-band Low-pass Filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

See Also   iirshiftc | zpkshiftc

Purpose   Automatic dynamic range scaling

Syntax   autoscale(hd,x)
                hnew = autoscale(hd,x)

Description autoscale(hd,x) provides dynamic range scaling for each node of the
                filter hd. This method runs signal x through hd in floating-point to
                simulate filtering. autoscale uses the maximum and minimum data
                obtained from that simulation at each filter node to set fraction lengths
                to cover the simulation full range and maximize the precision. Word
                lengths are not changed during autoscaling.

                hnew = autoscale(hd,x) If you request an output, autoscale returns
                a new filter with the scaled fraction lengths. The original filter is not
                changed.

                For introductory demonstrations of the automatic scale process, refer to
                the following demos in the toolbox:

                • Fixed-Point Scaling of an Elliptic IIR Filter

                • Floating-Point to Fixed-Point Conversion of IIR Filters

                • Floating-Point to Fixed-Point Conversion of FIR Filters

Examples   Demonstrate dynamic range scaling in a lattice ARMA filter:

```
hd = design(fdesign.bandpass,'ellip');
hd = convert(hd,'latticearma');
hd.arithmetic = 'fixed';
rng(4); x = rand(100,10); % Training input data.
hd(2) = autoscale(hd,x);
hfvt = fvtool(hd,'Analysis','magestimate','Showreference','off');
legend(hfvt, 'Before Autoscaling', 'After Autoscaling')
```

See Also   qreport

Purpose   Generate block from multirate or multistage filter

Syntax   block(hm)
                  block(hm,'propertyname1',propertyvalue1,'propertyname2',
                      propertyvalue2,...)

Description   block(hm) generates a DSP System Toolbox block equivalent to hm.

block(hm,'propertyname1',propertyvalue1,'propertyname2',propertyvalue2
generates a DSP System Toolbox block using the options specified in
the property name/property value pairs. The valid properties
and their values are

Property Name	Property Values	Description and Values
Destination	'current' (default), 'new', or *Subsystemname*.	Determine which Simulink model gets the block. Enter 'current', 'new', or specify the name of an existing subsystem with *subsystemname*. 'current' adds the block to your current Simulink model. Specifying 'new' opens a new model and adds the block. Specifying 'new' opens a new model and adds the block. If you provide the name of a subsystem in *subsystemname*, block adds the new block to your specified subsystem.
Blockname	'filter' (default)	Specify the name of the generated block. The name appears below the block in the model. When you do not specify a block name, the default is filter.

Property Name	Property Values	Description and Values
OverwriteBlock	'off' (default), or 'on'.	Tell block whether to overwrite an existing block of the same name, or create a new block. 'off' is the default setting—block does not overwrite existing blocks with matching names. Switching from 'off' to 'on' directs block to overwrite existing blocks.
MapStates	'off' (default), or 'on'.	Specify whether to apply the current filter states to the new block. This lets you save states from a filter object you may have used or configured in a specific way. The default setting of 'off' means the states are not transferred to the block. Choosing 'on' preserves the current filter states in the block.
Link2Obj	'off' (default), or 'on'.	Specify how to set the **Coefficient source** in the block mask. The default setting is 'off' and the **Coefficient source** is set to **Dialog parameters**. Setting Link2Obj to 'on' sets the **Coefficient source** to **Discrete-time filter object (DFILT)**. The Link2Obj and MapCoeffstoPorts cannot be simultaneously 'on'.

Property Name	Property Values	Description and Values
MapCoeffstoPorts	'off' (default) or 'on'	Specify whether to map the coefficients of the filter to the ports of the block. The Link2Obj and MapCoeffstoPorts cannot be simultaneously 'on'. See mfilt for a list of supported filter objects.
CoeffNames	{'Num'} (default FIR) {'Num','Den'} (default direct form IIR) {'Num','Den','g'} (default IIR SOS), {'K'} (default form lattice)	Specify the coefficient variable names as string variables in a cell array. MapCoeffsToPorts must be set to 'on' for this property to apply.
InputProcessing	'columns as channels' (default), 'elements as channels' , or 'inherited'	Specify sample-based (elements as channels) or frame-based (columns as channels) processing. The Inherited (this choice will be removed - see release notes) option will be removed in a future release.
RateOption	'enforce single rate' (default) or 'allow multirate'	Specify how the block adjusts the rate at the output to accommodate the reduced number of samples. This parameter applies only when InputProcessing is 'columns as channels'.

block

Using block to Realize Fixed-Point Multirate Filters

When the source filter hm is fixed-point, the input word and fraction lengths for the block are derived from the block input signal. The realization process issues a warning and ignores the input word and input fraction lengths that are part of the source filter object, choosing to inherit the settings from the input data. Other fixed-point properties map directly to settings for word and fraction length in the realized block.

Examples

Two examples of using block demonstrate the syntax capabilities. Both examples start from an mfilt object with interpolation factor of three. In the first example, use block with the default syntax, letting the function determine the block name and configuration.

```
hm = mfilt.firdecim(3);
```

Now use the default syntax to create a block.

```
block(hm);
```

In this second example, define the block name to meet your needs by using the property name/property value pair input arguments.

```
block(hm, 'blockname', 'firdecim');
```

The figure below shows the blocks in a Simulink model. When you try these examples, you see that the second block writes over the first block location. You can avoid this by moving the first block before you generate the second, always naming your block with the blockname property, or setting the Destination property to new which puts the filter block in a new Simulink model.

x[3n]

Filter

x[3n]

firdecim

See Also   `realizemdl`

How To   • "Realize Filters as Simulink Subsystem Blocks"

butter

Purpose	Butterworth IIR filter design using specification object
Syntax	hd = design(d,'butter') hd = design(d,'butter',designoption,value...)
Description	hd = design(d,'butter') designs a Butterworth IIR digital filter using the specifications supplied in the object d. hd = design(d,'butter',designoption,value...) returns a Butterworth IIR filter where you specify a design option and value. To determine the available design options, use designopts with the specification object and the design method as input arguments as shown. designopts(d,'method') For complete help about using butter, refer to the command line help system. For example, to get specific information about using butter with d, the specification object, enter the following at the MATLAB prompt. help(d,'butter')

Examples	Construct a default lowpass filter specification object and design a Butterworth filter:
	
	d = fdesign.lowpass;
	hd = design(d,'butter','matchexactly','stopband');
	
	Construct a highpass filter specification object and design a Butterworth filter:
	
	%Order 8 and 3 dB frequency 0.6*pi radians/sample
	d = fdesign.highpass('N,F3dB',8,.6);
	hd = design(d,'butter');
	fvtool(hd)

See Also   cheby1 | cheby2 | ellip

Purpose   Convert coupled allpass filter to transfer function form

Syntax   `[b,a]=ca2tf(d1,d2)`
                     `[b,a]=ca2tf(d1,d2,beta)`
                     `[b,a,bp]=ca2tf(d1,d2)`
                     `[b,a,bp]=ca2tf(d1,d2,beta)`

Description   `[b,a]=ca2tf(d1,d2)` returns the vector of coefficients b and the vector of
                     coefficients a corresponding to the numerator and the denominator of
                     the transfer function

$$H(z) = B(z)\,/\,A(z) = \frac{1}{2}\big[H1(z) + H2(z)\big]$$

d1 and d2 are real vectors corresponding to the denominators of the
allpass filters H1(z) and H2(z).

`[b,a]=ca2tf(d1,d2,beta)` where d1, d2 and beta are complex, returns
the vector of coefficients b and the vector of coefficients a corresponding
to the numerator and the denominator of the transfer function

$$H(z) = B(z)\,/\,A(z) = \frac{1}{2}\big[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z)\big]$$

`[b,a,bp]=ca2tf(d1,d2)`, where d1 and d2 are real, returns the vector
bp of real coefficients corresponding to the numerator of the power
complementary filter G(z)

$$G(z) = Bp(z)\,/\,A(z) = \frac{1}{2}\big[H1(z) - H2(z)\big]$$

`[b,a,bp]=ca2tf(d1,d2,beta)`, where d1, d2 and beta are complex,
returns the vector of coefficients bp of real or complex coefficients that
correspond to the numerator of the power complementary filter G(z)

$$G(z) = Bp(z)\,/\,A(z) = \frac{1}{2_j}\big[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z)\big]$$

Examples   Create a filter, convert the filter to coupled allpass form, and convert the
result back to the original structure (create the power complementary
filter as well).

```
[b,a]=cheby1(10,.5,.4);
[d1,d2,beta]=tf2ca(b,a);        % tf2ca returns the %
                                  denominators of the %
                                  allpasses.
[num,den,numpc]=ca2tf(d1,       % Reconstruct the original
d2,beta);                       % filter plus the power %
                                  complementary one.
[h,w,s]=freqz(num,den);
hpc = freqz(numpc,den);
s.plot = 'mag';
s.yunits = 'sq';
freqzplot([h hpc],w,s);         % Plot the mag response of
                                  the % original filter and
                                  the % power complementary
                                  one.
```

See Also   cl2tf | iirpowcomp | tf2ca | tf2cl

cheby1

Purpose	Chebyshev Type I filter using specification object

Syntax

```
hd = design(d,'cheby1')
hd = design(d,'cheby1',designoption,value,designoption,
value,...)
```

Description

`hd = design(d,'cheby1')` designs a type I Chebyshev IIR digital filter using the specifications supplied in the object `d`. Depending on the filter specification object, `cheby1` may or may not be a valid design. Use `designmethods` with the filter specification object to determine if a Chebyshev type I filter design is possible.

`hd = design(d,'cheby1',designoption,value,designoption, value,...)` returns a type I Chebyshev IIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `cheby1`, refer to the command line help system. For example, to get specific information about using `cheby1` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'cheby1')
```

Examples

These examples use filter specification objects to construct Chebyshev type I filters. In the first example, you use the `matchexactly` option to ensure the performance of the filter in the passband.

```
d = fdesign.lowpass; designopts(d,'cheby1');
hd = design(d,'cheby1','matchexactly','passband');
```

Specify the filter order, passband edge frequency, and the passband ripple to get the filter exactly as required.

```
d = fdesign.highpass('n,fp,ap',7,20,.4,50);
hd = design(d,'cheby1');
```

Use fvtool to view the resulting filter.

```
fvtool(hd)
```


By design, cheby1 returns filters that use second-order sections (SOS). For many applications, and for most fixed-point applications, SOS filters are particularly well-suited.

See Also   design | designmethods | fdesign

cheby2

Purpose	Chebyshev Type II filter using specification object

Syntax	`hd = design(d,'cheby2')`
	`hd = design(d,'cheby2',designoption,value,designoption,`
	`value,...)`

Description	`hd = design(d,'cheby2')` designs a Chebyshev II IIR digital filter using the specifications supplied in the object `d`.
	
	`hd = design(d,'cheby2',designoption,value,designoption,`
	`value,...)` returns a Chebyshev II IIR filter where you specify design options as input arguments.
	
	To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.
	
	`designopts(d,'method')`
	
	For complete help about using `cheby1`, refer to the command line help system. For example, to get specific information about using `cheby2` with `d`, the specification object, enter the following at the MATLAB prompt.
	
	`help(d,'cheby2')`

Examples	These examples use filter specification objects to construct Chebyshev type I filters. In the first example, you use the `matchexactly` option to ensure the performance of the filter in the passband.
	
	`d = fdesign.lowpass;`
	`hd = design(d,'cheby2','matchexactly','passband');`
	
	`cheby2` also designs highpass, bandpass, and bandstop filters. Here is a highpass filter where you specify the filter order, the stopband edge frequency. and the stopband attenuation to get the filter exactly as required.
	
	`d = fdesign.highpass('n,fst,ast',5,20,55,50);`

```
hd=design(d,'cheby2');
```

The Filter Visualization Tool shows the highpass filter meets the specifications.

```
fvtool(hd)
```


By design, cheby2 returns filters that use second-order sections. For many applications, and for most fixed-point applications, SOS filters are particularly well-suited for use.

See Also   butter | cheby1 | ellip

cl2tf

Purpose
Convert coupled allpass lattice to transfer function form

Syntax
```
[b,a] = cl2tf(k1,k2)
[b,a] = cl2tf(k1,k2,beta)
[b,a,bp] = cl2tf(k1,k2)
[b,a,bp] = cl2tf(k1,k2,beta)
```

Description
`[b,a] = cl2tf(k1,k2)` returns the numerator and denominator vectors of coefficients b and a corresponding to the transfer function

$$H(z) = B(z) / A(z) = \frac{1}{2}\big[H1(z) + H2(z)\big]$$

where *H1(z)* and *H2(z)* are the transfer functions of the allpass filters determined by `k1` and `k2`, and `k1` and `k2` are real vectors of reflection coefficients corresponding to allpass lattice structures.

`[b,a] = cl2tf(k1,k2,beta)` where `k1`, `k2` and `beta` are complex, returns the numerator and denominator vectors of coefficients b and a corresponding to the transfer function

$$H(z) = B(z) / A(z) = \frac{1}{2}\big[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z)\big]$$

`[b,a,bp] = cl2tf(k1,k2)` where `k1` and `k2` are real, returns the vector bp of real coefficients corresponding to the numerator of the power complementary filter *G(z)*

$$G(z) = Bp(z) / A(z) = \frac{1}{2}\big[H1(z) - H2(z)\big]$$

`[b,a,bp] = cl2tf(k1,k2,beta)` where `k1`, `k2` and `beta` are complex, returns the vector of coefficients bp of possibly complex coefficients corresponding to the numerator of the power complementary filter *G(z)*

$$G(z) = Bp(z) / A(z) = \frac{1}{2}_j \left[-(\bar{\beta}) \cdot H1(z) + \beta \cdot H2(z) \right]$$

Examples

```
[b,a]=cheby1(10,.5,.4);  %TF2CL returns the reflection coeffs
[k1,k2,beta]=tf2cl(b,a); % Reconstruct the original filter
% plus the power complementary one.
[num,den,numpc]=cl2tf(k1,k2,beta);
[h,w]=freqz(num,den); hpc = freqz(numpc,den);
% and the power complementary one.
subplot(211);
plot(w./pi,20*log10(abs(h)),'k'); hold on; grid on;
plot(w./pi,20*log10(abs(hpc)),'b');
xlabel('Normalized Frequency (x \pi radians/sample)');
ylabel('dB');
legend('Original Filter','Power Complementary Filter',...
    'Location','best');
subplot(212);
plot(w./pi,unwrap(angle(h)),'k'); hold on; grid on;
xlabel('Normalized Frequency (x \pi radians/sample)');
ylabel('Phase (radians)');
plot(w./pi,unwrap(angle(hpc)),'b');
```

See Also   tf2cl | tf2ca | ca2tf | tf2latc | latc2tf | iirpowcomp

Purpose	Coefficients for filters

Syntax

```
s = coeffs(ha)
s = coeffs(hd)
s = coeffs(hm)
s = coeffs(hs)
s = coeffs(hs,Name,Value)
```

Description   s = coeffs(ha) returns a structure containing the coefficients of the adaptive filter ha.

s = coeffs(hd) returns a structure containing the coefficients of the discrete-time filter hd.

s = coeffs(hm) returns a structure containing the coefficients of the multirate filter hm.

s = coeffs(hs) returns filter coefficients for the filter System object hs.

s = coeffs(hs,Name,Value) returns filter coefficients for the filter System object hs with additional options specified by one or more Name,Value pair arguments.

Input Arguments

ha

Adaptive adaptfilt filter object.

hd

Discrete-time dfilt filter object.

hm

Multirate mfilt filter object. CIC-based filters do not have coefficients, so the function does not support CIC filter structures such as mfilt.cicdecim.

hs

coeffs

Filter System object. The function supports the following filter structures.

- `dsp.FIRFilter`

- `dsp.BiquadFilter`

- `dsp.FIRInterpolator`

- `dsp.FIRDecimator`

- `dsp.FIRRateConverter`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Arithmetic

`double' | 'single' | 'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'`, the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

Output Arguments

s

Structure with a single field, Numerator, containing filter coefficients. For adaptive filters, s contains the instantaneous frequency response available at the time you use the function.

Examples

coeffs works the same way for all filters. This example uses a multirate filter hm to demonstrate the function.

```
hm=mfilt.firdecim(3);
s=coeffs(hm);
stem(s.Numerator)
```


The filter coefficients may be extracted by typing s.Numerator at the command prompt.

See Also   adaptfilt | freqz | grpdelay | impz | info | phasez | stepz | zerophase | zplane

coeread

Purpose	Read Xilinx COE file
Syntax	hd = coeread(filename)
Description	hd = coeread(filename) extracts the Distributed Arithmetic FIR filter coefficients defined in the XILINX CORE Generator .COE file specified by filename. It returns a dfilt object, the fixed-point filter hd. If you do not provide the file type extension .coe with the filename, the function assumes the .coe extension.
See Also	coewrite | dfilt | dfilt.dffir

Purpose   Write Xilinx COE file

Syntax
```
coewrite(hd)
coewrite(hd,radix)
coewrite(...,filename)
```

Description   coewrite(hd) writes a XILINX Distributed Arithmetic FIR filter
coefficient .COE file which can be loaded into the XILINX CORE
Generator. The coefficients are extracted from the fixed-point dfilt
object hd. Your fixed-point filter must be a direct form FIR structure
dfilt object with one section and whose Arithmetic property is set
to fixed. You cannot export single-precision, double-precision, or
floating-point filters as .coe files, nor multiple-section filters. To enable
you to provide a name for the file, coewrite displays a dialog box where
you fill in the file name. If you do not specify the name of the output
file, the default file name is untitled.coe.

coewrite(hd,radix) indicates the radix (number base) used to specify
the FIR filter coefficients. Valid radix values are 2 for binary, 10 for
decimal, and 16 for hexadecimal (default).

coewrite(...,filename) writes a XILINX.COE file to filename. If
you omit the file extension, coewrite adds the .coe extension to the
name of the file.

The coewrite function always generates the XILINX.COE file in your
current folder. To use this function, you must have write permission in
your current folder.

Examples   coewrite generates an ASCII text file that contains the filter
coefficients in a format the XILINX CORE Generator can read and load.
In this example, you create a 30th-order fixed-point filter and generate
the .coe file that includes the filter coefficients as well as associated
information about the filter.

```
b = firceqrip(30,0.4,[0.05 0.03]); hq = dfilt.dffir(b);
set(hq,'arithmetic','fixed'); coewrite(hq,10,'mycoefile');
```

The coewrite function generates the output file, mycoefile.coe, in your current folder. The .coe file contains the radix, coefficient width, and filter coefficients. The file reports the filter coefficients in column-major order. The radix, coefficient width, and filter coefficients are the minimum set of data needed in a .coe file.

See Also   coeread | dfilt | dfilt.dffir

Purpose   Constrain coefficient wordlength

Syntax

```
Hq = constraincoeffwl(Hd,wordlength)
Hq = constraincoeffwl(Hd,wordlength,'Ntrials',N)
Hq = constraincoeffwl(Hd,wordlength,...,'NoiseShaping',
    NSFlag)
Hq = constraincoeffwl(Hd,wordlength,...,'Apasstol',Apasstol)
Hq = constraincoeffwl(Hd,wordlength,...,'Astoptol',Astoptol)
```

Description   `Hq = constraincoeffwl(Hd,wordlength)` returns a fixed-point filter `Hq` meeting the design specifications of the single-stage or multistage FIR filter object `Hd` with a wordlength of at most `wordlength` bits. For multistage filters, `wordlength` can either be a scalar or vector. If `wordlength` is a scalar, the same wordlength is used for all stages. If `wordlength` is a vector, each stage uses the corresponding element in the vector. The vector length must equal the number of stages. `Hd` must be generated using `fdesign` and `design`. `constraincoeffwl` uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator `rand`

`Hq = constraincoeffwl(Hd,wordlength,'Ntrials',N)` specifies the number of Monte Carlo trials to use. `Hq` is first filter among the trials to meet the specifications in `Hd` with a wordlength of at most `wordlength`.

`Hq = constraincoeffwl(Hd,wordlength,...,'NoiseShaping',NSFlag)` enables or disables the stochastic noise-shaping procedure in the constraint of the wordlength. By default `NSFlag` is `true`. Setting `NSFlag` to `false` constrains the wordlength without using noise-shaping.

`Hq = constraincoeffwl(Hd,wordlength,...,'Apasstol',Apasstol)` specifies the passband ripple tolerance in dB. `'Apasstol'` defaults to `1e-4`.

`Hq = constraincoeffwl(Hd,wordlength,...,'Astoptol',Astoptol)` specifies the stopband tolerance in dB. `'Astoptol'` defaults to `1e-2`

constraincoeffwl

You must have the Fixed-Point Toolbox software installed to use this function.

Examples   Design fixed-point filter with a wordlength of at most 11 bits:

```
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',.4,.5,1,60);
Hd = design(Hf,'equiripple'); % 43 coefficients
Hq = constraincoeffwl(Hd,11); % 45 11-bit coefficients
hfvt = fvtool(Hd,Hq,'showreference','off');
legend(hfvt,'Floating-point Filter','Fixed-point Filter');
```


See Also   design | fdesign | maximizestopband | minimizecoeffwl | measure
| rand

Tutorials   • "Fixed-Point Data Types"

Purpose   Convert filter structure of discrete-time or multirate filter

Syntax   ```
                hq = convert(hq,newstruct)
                hm = convert(hm,newstruct)
                ```
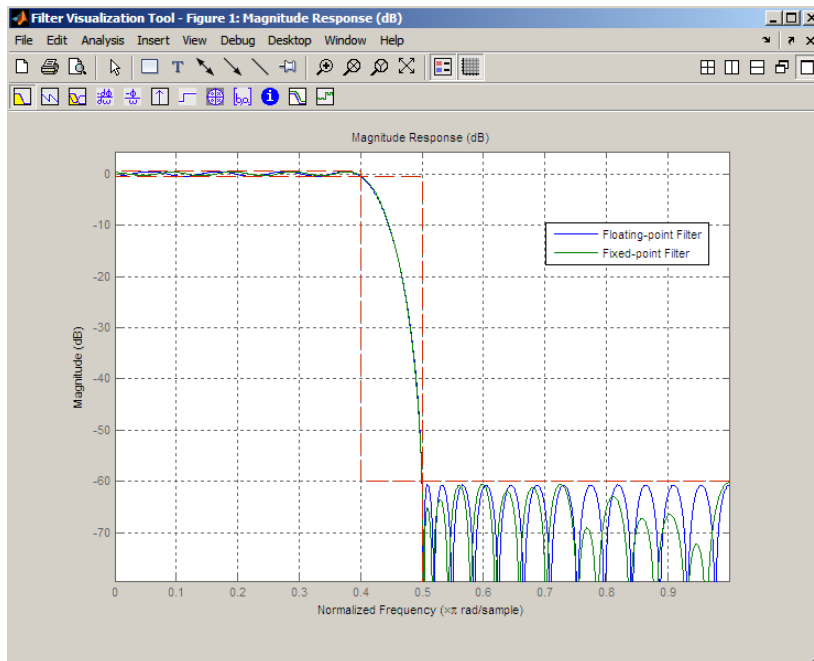
Description   **Discrete-Time Filters**

hq = convert(hq,newstruct) returns a quantized filter whose structure has been transformed to the filter structure specified by string newstruct. You can enter any one of the following quantized filter structures:

- 'antisymmetricfir': Antisymmetric finite impulse response (FIR)

- 'df1': Direct form I

- 'df1t': Direct form I transposed

- 'df1sos': Direct-Form I, Second-Order Sections

- 'df1tsos': Direct-Form I Transposed, Second-Order Sections

- 'df2': Direct form II

- 'df2t': Direct form II transposed. Default filter structure

- 'df2sos': Direct-Form II, Second-Order Sections

- 'df2tsos': Direct-Form II Transposed, Second-Order Sections

- 'dffir': FIR

- 'dffirt': Direct form FIR transposed

- 'latcallpass': Lattice allpass

- 'latticeca': Lattice coupled-allpass

- 'latticecapc': Lattice coupled-allpass power-complementary

- 'latticear': Lattice autoregressive (AR)

- 'latticemamax': Lattice moving average (MA) maximum phase

- 'latticemamin': Lattice moving average (MA) minimum phase

- `'latticearma'`: Lattice ARMA
- `'statespace'`: Single-input/single-output state-space
- `'symmetricfir'`: Symmetric FIR. Even and odd forms

All filters can be converted to the following structures:

- `'df1'`: Direct form I
- `'df1t'`: Direct form I transposed
- `'df1sos'`: Direct-Form I, Second-Order Sections
- `'df1tsos'`: Direct-Form I Transposed, Second-Order Sections
- `'df2'`: Direct form II
- `'df2t'`: Direct form II transposed. Default filter structure
- `'df2sos'`: Direct-Form II, Second-Order Sections
- `'df2tsos'`: Direct-Form II Transposed, Second-Order Sections
- `'statespace'`: Single-input/single-output state-space
- `'symmetricfir'`: Symmetric FIR. Even and odd forms

For the following filter classes, you can specify other conversions as well:

- Minimum phase FIR filters can be converted to `latticemamin`
- Maximum phase FIR filters can be converted to `latticemamax`
- Allpass filters can be converted to `latcallpass`

`convert` generates an error when you specify a conversion that is not possible.

Multirate Filters

`hm = convert(hm,newstruct)` returns a multirate filter whose structure has been transformed to the filter structure specified by string `newstruct`. You can enter any one of the following multirate filter structures, defined by the strings shown, for `newstruct`:

Cascaded Integrator-Comb Structure

- `cicdecim` — CIC-based decimator

- `cicinterp` — CIC-based interpolator

FIR Structures

- `firdecim` — FIR decimator

- `firtdecim` — transposed FIR decimator

- `firfracdecim` — FIR fractional decimator

- `firinterp` — FIR interpolator

- `firfracinterp` — FIR fractional interpolator

- `firsrc` — FIR sample rate change filter

- `firholdinterp` — FIR interpolator that uses hold interpolation between input samples

- `firlinearinterp` — FIR interpolator that uses linear interpolation between input samples

- `fftfirinterp` — FFT-based FIR interpolator

You cannot convert between the FIR and CIC structures.

Examples   `[b,a]=ellip(5,3,40,.7); hq = dfilt.df2t(b,a);`
`hq2 = convert(hq,'df1');`

For an example of changing the structure of a multirate filter, try the following conversion from a CIC interpolator to a CIC interpolator with zero latency.

`hm = mfilt.cicinterp(2,2,3,8,8);`

> **Note** The above example will generate a warning:
>
> ```
> Warning: Using reference filter for structure conversion.
> Fixed-point attributes will not be converted.
> ```
>
> Since CIC interpolators only use fixed-point arithmetic, the user may disregard this warning. The fixed-point structure will not be lost on conversion.

See Also   mfilt | dfilt

Purpose   Estimate cost of discrete-time or multirate filter

Syntax   ```
c = cost(hd)
c = cost(hm)
c = cost(hs)
c = cost(hs,Name,Value)
```

**Description**  `c = cost(hd)` returns a cost estimate `c` for the discrete-time filter `hd`.

`c = cost(hm)` return a cost estimate `c` for the multirate filter `hm`.

`c = cost(hs)` returns a cost estimate `c` for the filter System object `hs`.

`c = cost(hs,Name,Value)` returns a cost estimate `c` for the filter System object `hs` with additional options specified by one or more `Name,Value` pair arguments.

**Input Arguments**

**hd**

Discrete-time `dfilt` filter object.

**hm**

Multirate `mfilt` filter object.

**hs**

Filter System object. The function supports the following filter structures.

- `dsp.FIRFilter`
- `dsp.BiquadFilter`
- `dsp.FIRInterpolator`
- `dsp.CICInterpolator`
- `dsp.FIRDecimator`
- `dsp.CICDecimator`
- `dsp.FIRRateConverter`

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### Arithmetic
`` `double' | 'single' | 'fixed' ``

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'`, the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

### Details for Fixed-Point Arithmetic

| System Object State | Coefficient Data Type | Rule |
| --- | --- | --- |
| Unlocked | `'Same as input'` | The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Unlocked | `'Custom'` | The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property. |

| System Object State | Coefficient Data Type | Rule |
|---|---|---|
| Locked | `'Same as input'` | When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Locked | `'Custom'` | The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property. |

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

**Output Arguments**

`c`

Cost estimate. `c` has the following fields.

**cost**

| Estimated Value | Field Name | Description |
|---|---|---|
| Number of Multiplications | nmult | Number of multiplications during the filter run. nmult ignores multiplications by -1, 0, and 1 in the total multiple. |
| Number of Additions | nadd | Number of additions during the filter run. |
| Number of States | nstates | Number of states the filter uses. |
| Multiplications per Input Sample | multperinputsample | Number of multiplication operations performed for each input sample |
| Additions per Input Sample | addperinputsample | Number of addition operations performed for each input sample |

**Examples**   These examples show you the cost method applied to dfilt and mfilt objects.

```
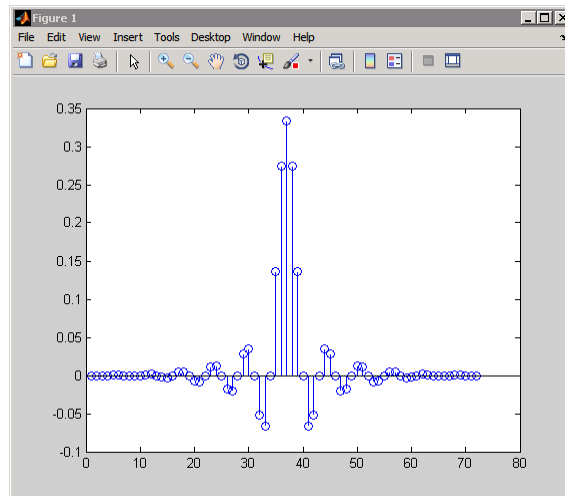hd = design(fdesign.lowpass);
c = cost(hd);
```

When you are using a multirate filter object, cost works the same way.

```
d = fdesign.decimator(4,'cic');
hm = design(d,'multisection');
```

See Also   qreport

Purpose   Vector of SOS filters for cumulative sections

Syntax
```
h = cumsec(hd)
h = cumsec(hd,indices)
h = cumsec(hd,indices,secondary)
cumsec(hd,...)
h = cumsec(hs)
h = cumsec(hs,Name,Value)
```

Description   `h = cumsec(hd)` returns a vector `h` of SOS filter objects with the cumulative sections. Each element in `h` is a filter with the structure of the original filter. The first element is the first filter section of `hd`. The second element of `h` is a filter that represents the combination of the first and second sections of `hd`. The third element of `h` is a filter which combines sections 1, 2, and 3 of `hd`. This pattern continues until the final element of `h` contains all the sections of `hd` and should be identical to `hd`.

`h = cumsec(hd,indices)` returns a vector `h` of SOS filter objects whose indices into the original filter are in the vector `indices`.

`h = cumsec(hd,indices,secondary)` uses the secondary scaling points `secondary` in the sections to determine where the sections should be split.

`cumsec(hd,...)` uses FVTool to plot the magnitude response of the cumulative sections.

`h = cumsec(hs)` returns the cumulative sections of the `dsp.BiquadFilter` filter System object `hs`. You can also use the optional input arguments `indices` and `secondary` with this syntax. You can also omit the output argument `h` to use FVTool to plot the magnitude response of the cumulative sections.

`h = cumsec(hs,Name,Value)` returns the cumulative sections of the filter System object `hs` with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments	**hd**

Discrete-time `dfilt.df1sos`, `dfilt.df2tsos`, `dfilt.df2sos`, or `dfilt.df1tsos`filter object.

hs

`dsp.BiquadFilter` filter System object.

indices

Filter indices. Use `indices` to specify the filter sections `cumsec` uses to compute the cumulative responses.

secondary

This option applies only when `hd` is a `df2sos` and `df1tsos` filter. For these second-order section structures, the secondary scaling points refer to the scaling locations between the recursive and the nonrecursive parts of the section (the "middle" of the section). Argument `secondary` accepts either `true` or `false`. By default, `secondary` is `false`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Arithmetic

`double' | 'single' | 'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

Examples   To demonstrate how `cumsec` works, this example plots the relative responses of the sections of a sixth-order filter SOS filter with three sections. Each curve adds one more section to form the filter response.

```
hs = fdesign.lowpass('n,fc',6,.4); hd = butter(hs);
h = cumsec(hd); hfvt = fvtool(h);
legend(hfvt,'First Section','First Two Sections','Overall  Filter');
```


See Also   scale | scalecheck

Purpose   Undo filter coefficient and gain changes caused by normalize

Syntax   denormalize(hq)

Description denormalize(hq) reverses the coefficient changes you make when you
use normalize with hq. The filter coefficients do not change if you call
denormalize(hq) before you use normalize(hq). Calling denormalize
more than once on a filter does not change the coefficients after the
first denormalize call.

Examples   Make a quantized filter hq and normalize the filter coefficients. After
normalizing the coefficients, restore them to their original values by
reversing the effects of the normalize function.

```
d=fdesign.highpass('n,F3dB',14,0.45);
hd =design(d,'butter');
hd.arithmetic='fixed';
normalize(hd)
NormSOSMatrix = hd.sosMatrix;
denormalize(hd)
eqSOSMatrices = isequal(NormSOSMatrix,hd.sosMatrix);
% returns a 0
```

See Also   normalize

design

Purpose	Apply design method to filter specification object
Syntax	H = design(D) H = design(D,METHOD) H = design(D,METHOD,PARAM1,VALUE1,PARAM2,VALUE2,...) H = design(D,METHOD,OPTS) Hs = design(D,...,'SystemObject',*sysobjflag*)

Description   H = design(D) uses the filter specifications object D to generate a filter H. When you do not provide a design method as an input argument, design uses a default design method. Use designmethods(D,'default') to see the default design method for your filter specifications object.

H = design(D,METHOD) forces the design method specified by the string METHOD. METHOD must be one of the strings returned by designmethods. Use designmethods(D,'default') to determine which algorithm is used by default.

The design method you provide as the designmethod input argument must be one of the methods returned by

designmethods(d)

To help you design filters more quickly, the input argument METHOD accepts a variety of special keywords that force design to behave in different ways. The following table presents the keywords you can use for METHOD and how design responds to the keyword.

Designmethod Keyword	Description of the design Response
'FIR'	Forces design to produce an FIR filter. When no FIR design method exists for object D, design returns an error.
'IIR'	Forces design to produce an IIR filter. When no IIR design method exists for object D, design returns an error.

Designmethod Keyword	Description of the design Response
`'ALLFIR'`	Produces filters from every applicable FIR design method for the specifications in D, one filter for each design method. As a result, `design` returns multiple filters in the output object.
`'ALLIIR'`	Produces filters from every applicable IIR design method for the specifications in D, one filter for each design method. As a result, `design` returns multiple filters in the output object.
`'ALL'`	Designs filters using all applicable design methods for the specifications object D. As a result, `design` returns multiple filters, one for each design method. `design` uses the design methods in the order that `designmethods(D)` returns them.

Keywords are not case sensitive

When `design` returns multiple filters in the output object, use indexing to see the individual filters. For example, to see the third filter in H, enter

`H(3)`

`H = design(D,METHOD,PARAM1,VALUE1,PARAM2,VALUE2,...)` specifies design-method options. Use `help(D,METHOD)` for complete information on which design-method-specific options are available. You can also use `designopts(D,METHOD)` for a less-detailed listing of the design-method-specific options.

`H = design(D,METHOD,OPTS)` specifies design-method options using the structure `OPTS`. `OPTS` is usually obtained from `designopts` and then specified as an input to `design`. Use `help(D,METHOD)` for more information on optional inputs.

`Hs = design(D,...,'SystemObject',`*sysobjflag*`)` uses the filter specifications object D to generate a filter System object Hs when

sysobjflag is `true`. To generate System objects, you must have the DSP System Toolbox product installed. When *sysobjflag* is `false`, the function generates a `dfilt` or `mfilt` object H, as described previously. Design methods and design options for filter System objects are not necessarily the same as those for `dfilt` and `mfilt` objects. To check design methods for System objects, use `designmethods` with the `'SystemObject'`,*sysobjflag* syntax.

If you are specifying design-method-specific options using OPTS, you can also set OPTS.SystemObject to `true` instead of calling `design` with the `'SystemObject'`,*sysobjflag* syntax.

Examples

Design an FIR equiripple lowpass filter. The passband edge frequency is 0.2π radians/sample, and the stopband edge frequency is 0.25π radians/sample. The passband ripple is 0.5 dB, and the stopband attenuation is 40 dB.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,0.5,40);
H = design(D); % Uses the default equiripple method.
```

If you have the DSP System Toolbox software installed, you can design a minimum-phase FIR equripple filter. Design a minimum–phase filter and compare the pole-zero plots of the original and minimum-phase designs.

```
Hmin = design(D,'equiripple','MinPhase',true);
hfvt = fvtool([H Hmin],'analysis','polezero');
legend(hfvt,'Original Design','Minimum Phase Design');
```

Design a Butterworth lowpass filter. The passband edge frequency is 0.2π radians/sample, and the stopband edge frequency is 0.25π radians/sample. The passband ripple is 0.5 dB, and the stopband attenuation is 40 dB. Obtain help on the design options specific to the Butterworth design method. Design the filter with the "MatchExactly' option set to 'Passband'.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,0.5,40);
% Query design-method-specific options
```

```
help(D,'butter')
% Match passband exactly
H = design(D,'butter','MatchExactly','passband');
```

If you have the DSP System Toolbox software, you can specify the *P*-th norm scaling on the second-order sections. Use L-infinity norm scaling in the time domain.

```
H = design(D,'butter','MatchExactly','passband','SOSScaleNorm','linf')
```

If you have the DSP System Toolbox software, you can create a filter System object.

```
Hs = design(D,'SystemObject',true);
```

See Also   designmethods | designopts

designmethods

Purpose　　　Methods available for designing filter from specification object

Syntax　　　　
```
M = designmethods(D)
M = designmethods(D,'default')
M = designmethods(D,TYPE)
M = designmethods(D,'full')
Ms = designmethods(D,...,'SystemObject',sysobjflag)
```

Description　　`M = designmethods(D)` returns the available design methods for the filter specification object, `D`, and the current value of the `Specification` property.

`M = designmethods(D,'default')` returns the default design method for the filter specification object `D` and the current value of the `Specification` property.

`M = designmethods(D,TYPE)` returns either the TYPE design methods that apply to D. TYPE can be either `'FIR'` or `'IIR'`.

`M = designmethods(D,'full')` returns the full name for each of the available design methods. For example, `designmethods` with the `'full'` argument returns `Butterworth` for the `butter` method.

`Ms = designmethods(D,...,'SystemObject',sysobjflag)` returns the available design methods for designing filter System objects when *sysobjflag* is `true`. To use System objects, you must have the DSP System Toolbox product installed. When *sysobjflag* is `false`, the function checks methods for creating `dfilt` and `mfilt` objects, as described previously. Design methods and design options for filter System objects are not necessarily the same as those for `dfilt` and `mfilt` objects.

Examples　　　Construct a lowpass filter specification object and determine the valid design methods. Obtain detailed command line help on the Chebyshev type I design method.

```
D =fdesign.lowpass('Fp,Fst,Ap,Ast',500,600,0.5,60,1e4);
M = designmethods(D)
help(D,M{2})
```

The last line of the example is equivalent to `help(D,'cheby1')`.

If you have DSP System Toolbox software installed, use the `'SystemObject',`*sysobjflag* syntax to return design methods for a filter System object:

```
Ms = designmethods(D,'SystemObject',true);
```

See Also   design | designopts | fdesign

designopts

Purpose	Valid input arguments and values for specification object and method
Syntax	OPTS = designopts(D,METHOD)
Description	OPTS = designopts(D,METHOD) returns a structure array with the default design parameters used by the design method METHOD. METHOD must be one of the strings returned by designmethods.
	Use help(D,METHOD) to get a description of the design parameters.
	If you have DSP System Toolbox software installed, OPTS has the SystemObject property if at least one of the structures available for that design method is supported by System objects. However, not all structures for that design method are supported by System objects.
Examples	Create a lowpass filter with a numerator and denominator order of 10 and a 3-dB frequency of 0.2π radians/sample. Obtain the default design parameters for a Butterworth design, and test whether the filter structure is a direct-form II biquad.

```
D = fdesign.lowpass('Nb,Na,F3dB',10,10,0.2);
OPTS = designopts(D,'butter');
if (OPTS.FilterStructure == 'df2sos')
    fprintf('The default filter structure is Direct-Form II\n');
    fprintf('with second-order sections.\n');
end
```

If you have DSP System Toolbox software installed, OPTS has the SystemObject property.

See Also design | designmethods | fdesign | validstructures

Purpose   Discrete-time filter

Syntax   hd = dfilt.*structure*(input1,...)
                 hd = [dfilt.*structure*(input1,...), dfilt.*structure*(input1,
                    ...),...]
                 hd = design(d,'designmethod')

Description   hd = dfilt.*structure*(input1,...) returns a discrete-time filter, hd,
                 of type *structure*. Each structure takes one or more inputs. When you
                 specify a dfilt.*structure* with no inputs, a default filter is created.

Note You must use a *structure* with dfilt.

---

hd = [dfilt.*structure*(input1,...),
dfilt.*structure*(input1,...),...] returns a vector containing
dfilt filters.

Structures

Structures for dfilt.structure specify the type of filter structure.
Available types of structures for dfilt are shown below.

dfilt.structure	Description	Coefficient Mapping Support in realizemdl
dfilt.allpass	Allpass filter	Supported
dfilt.cascadeallpass	Cascade of allpass filter sections	Supported
dfilt.cascadewdfallpass	Cascade of allpass wave digital filters	Supported
dfilt.delay	Delay	Not supported
dfilt.df1	Direct-form I	Supported
dfilt.df1sos	Direct-form I, second-order sections	Supported

dfilt.structure	Description	Coefficient Mapping Support in realizemdl
dfilt.df1t	Direct-form I transposed	Supported
dfilt.df1tsos	Direct-form I transposed, second-order sections	Supported
dfilt.df2	Direct-form II	Supported
dfilt.df2sos	Direct-form II, second-order sections	Supported
dfilt.df2t	Direct-form II transposed	Supported
dfilt.df2tsos	Direct-form II transposed, second-order sections	Supported
dfilt.dffir	Direct-form FIR	Supported
dfilt.dffirt	Direct-form FIR transposed	Supported
dfilt.dfsymfir	Direct-form symmetric FIR	Supported
dfilt.dfasymfir	Direct-form antisymmetric FIR	Supported
dfilt.farrowfd	Generic fractional delay Farrow filter	Supported
dfilt.farrowlinearfd	Linear fractional delay Farrow filter	Not supported
dfilt.fftfir	Overlap-add FIR	Not supported
dfilt.latticeallpass	Lattice allpass	Supported
dfilt.latticear	Lattice autoregressive (AR)	Supported
dfilt.latticearma	Lattice autoregressive moving-average (ARMA)	Supported
dfilt.latticemamax	Lattice moving-average (MA) for maximum phase	Supported
dfilt.latticemamin	Lattice moving-average (MA) for minimum phase	Supported

dfilt.structure	Description	Coefficient Mapping Support in realizemdl
dfilt.calattice	Coupled, allpass lattice	Supported
dfilt.calatticepc	Coupled, allpass lattice with power complementary output	Supported
dfilt.statespace	State-space	Supported
dfilt.scalar	Scalar gain object	Supported
dfilt.wdfallpass	Allpass wave digital filter object	Supported
dfilt.cascade	Filters arranged in series	Supported
dfilt.parallel	Filters arranged in parallel	Supported

For more information on each structure, refer to its reference page.

hd = design(d,'designmethod') returns the dfilt object hd resulting from the filter specification object d and the design method you specify in *designmethod*. When you omit the designmethod argument, design uses the default design method to construct a filter from the object d.

With this syntax, you design filters by:

1 Specifying the filter specifications, such as the response shape (perhaps highpass) and details (passband edges and attenuation).

2 Selecting a method (such as equiripple) to design the filter.

3 Applying the method to the specifications object with design(d,'designmethod).

Using the specification-based technique can be more effective than the coefficient-based filter design techniques.

Design Methods for Design Syntax

When you use the hd = design(d,'*designmethod*') syntax, you have a range of design methods available depending on d, the filter

specification object. The next table lists all of the design methods in the toolbox.

Design Method String	Filter Design Result
butter	Butterworth IIR
cheby1	Chebyshev Type I IIR
cheby2	Chebyshev Type II IIR
ellip	Elliptic IIR
equiripple	Equiripple with the same ripple in the pass and stopbands
firls	Least-squares FIR
freqsamp	Frequency-Sampled FIR
ifir	Interpolated FIR
iirlpnorm	Least Pth norm IIR
iirls	Least-Squares IIR
kaiserwin	Kaiser-windowed FIR
lagrange	Fractional delay filter
multistage	Multistage FIR
window	Windowed FIR

As the specifications object d changes, the available methods for designing filters from d also change. For instance, if d is a lowpass filter with the default specification 'Fp,Fst,Ap,Ast', the applicable methods are:

```
% Create an object to design a lowpass filter.
d=fdesign.lowpass;
designmethods(d) % What design methods apply to object d?
```

If you change the specification string to `'N,F3dB'`, the available design methods change:

```
d=fdesign.lowpass('N,F3dB');
designmethods(d)
```

Analysis Methods

Methods provide ways of performing functions directly on your `dfilt` object without having to specify the filter parameters again. You can apply these methods directly on the variable you assigned to your `dfilt` object.

For example, if you create a `dfilt` object, `hd`, you can check whether it has linear phase with `islinphase(hd)`, view its frequency response plot with `fvtool(hd),` or obtain its frequency response values with `h = freqz(hd)`. You can use all of the methods described here in this way.

Note If your variable `hd` is a 1-D array of `dfilt` filters, the method is applied to each object in the array. Only `freqz`, `grpdelay`, `impz`, `is*`, `order`, and `stepz` methods can be applied to arrays. The `zplane` method can be applied to an array only if `zplane` is used without outputs.

Some of the methods listed here have the same name as functions in Signal Processing Toolbox software. They behave similarly.

Method	Description
addstage	Adds a stage to a cascade or parallel object, where a stage is a separate, modular filter. Refer to dfilt.cascade and dfilt.parallel.
block	block(hd) creates a block of the dfilt object. The block method can specify these properties and values:
	'Destination' indicates where to place the block. 'Current' places the block in the current Simulink model. 'New' creates a new model. Default value is 'Current'.
	'Blockname' assigns the entered string to the block name. Default name is 'Filter'.
	'OverwriteBlock' indicates whether to overwrite the block generated by the block method ('on') and defined by Blockame. Default is 'off'.
	'MapStates' specifies initial conditions in the block ('on'). Default is 'off'. Refer to "Using Filter States" in Signal Processing Toolbox documentation.
cascade	Returns the series combination of two dfilt objects. Refer to dfilt.cascade.
coeffs	Returns the filter coefficients in a structure containing fields that use the same property names as those in the original dfilt.
convert	Converts a dfilt object from one filter structure, to another filter structure.

Method	Description
fcfwrite	Writes a filter coefficient ASCII file. The file can contain a single filter or a vector of objects. The file can contain multirate filters (mfilt) or adaptive filters (adaptfilt). The default file name is untitled.fcf.
	fcfwrite(hd,filename) writes to a disk file named filename in the current working folder. The .fcf extension is added automatically.
	fcfwrite(...,fmt) writes the coefficients in the format fmt, where valid fmt strings are:
	'hex' for hexadecimal
	'dec' for decimal
	'bin' for binary representation
fftcoeffs	Returns the frequency-domain coefficients used when filtering with a dfilt.fftfir
filter	Performs filtering using the dfilt object.
firtype	Returns the type (1-4) of a linear phase FIR filter.
freqz	Plots the frequency response in fvtool. Unlike the freqz function, this dfilt freqz method has a default length of 8192.
grpdelay	Plots the group delay in fvtool.
impz	Plots the impulse response in fvtool.
impzlength	Returns the length of the impulse response.
info	Displays dfilt information, such as filter structure, length, stability, linear phase, and, when appropriate, lattice and ladder length.

Method	Description
isallpass	Returns a logical 1 (i.e., true) if the dfilt object in an allpass filter or a logical 0 (i.e., false) if it is not.
iscascade	Returns a logical 1 if the dfilt object is cascaded or a logical 0 if it is not.
isfir	Returns a logical 1 if the dfilt object has finite impulse response (FIR) or a logical 0 if it does not.
islinphase	Returns a logical 1 if the dfilt object is linear phase or a logical 0 if it is not.
ismaxphase	Returns a logical 1 if the dfilt object is maximum-phase or a logical 0 if it is not.
isminphase	Returns a logical 1 if the dfilt object is minimum-phase or a logical 0 if it is not.
isparallel	Returns a logical 1 if the dfilt object has parallel stages or a logical 0 if it does not.
isreal	Returns a logical 1 if the dfilt object has real-valued coefficients or a logical 0 if it does not.
isscalar	Returns a logical 1 if the dfilt object is a scalar or a logical 0 if it is not scalar.
issos	Returns a logical 1 if the dfilt object has second-order sections or a logical 0 if it does not.
isstable	Returns a logical 1 if the dfilt object is stable or a logical 0 if it are not.

Method	Description
nsections	Returns the number of sections in a second-order sections filter. If a multistage filter contains stages with multiple sections, using nsections returns the total number of sections in all the stages (a stage with a single section returns 1).
nstages	Returns the number of stages of the filter, where a stage is a separate, modular filter.
nstates	Returns the number of states for an object.
order	Returns the filter order. If hd is a single-stage filter, the order is given by the number of delays needed for a minimum realization of the filter. If hd has multiple stages, the order is given by the number of delays needed for a minimum realization of the overall filter.
parallel	Returns the parallel combination of two dfilt filters. Refer to dfilt.parallel.
phasez	Plots the phase response in fvtool.
realizemdl	(Available only with Simulink.) realizemdl(hd) creates a Simulink model containing a subsystem block realization of your dfilt. realizemdl(hd,p1,v1,p2,v2,...) creates the block using the properties p1, p2,... and values v1, v2,... specified. The following properties are available: 'Blockname' specifies the name of the block. The default value is 'Filter'. 'Destination' specifies whether to add the block to a current Simulink model or create a

Method	Description
	new model. Valid values are `'Current'` and `'New'`.
	`'OverwriteBlock'` specifies whether to overwrite an existing block that was created by realizemdl or create a new block. Valid values are `'on'` and `'off'`. Only blocks created by realizemdl are overwritten.
	The following properties optimize the block structure. Specifying `'on'` turns the optimization on and `'off'` creates the block without optimization. The default for each block is `'off'`.
	`'OptimizeZeros'` removes zero-gain blocks.
	`'OptimizeOnes'` replaces unity-gain blocks with a direct connection.
	`'OptimizeNegOnes'` replaces negative unity-gain blocks with a sign change at the nearest summation block.
	`'OptimizeDelayChains'` replaces delay chains made up of n unit delays with a single delay by n.
removestage	Removes a stage from a cascade or parallel dfilt. Refer to dfilt.cascade and dfilt.parallel.
setstage	Overwrites a stage of a cascade or parallel dfilt. Refer to dfilt.cascade and dfilt.parallel.

Method	Description
sos	Converts the dfilt to a second-order sections dfilt. If hd has a single section, the returned filter has the same class.
	sos(hd,flag) specifies the ordering of the second-order sections. If flag='UP', the first row contains the poles closest to the origin, and the last row contains the poles closest to the unit circle. If flag='down', the sections are ordered in the opposite direction. The zeros are always paired with the poles closest to them.
	sos(hd,flag,scale) specifies the scaling of the gain and the numerator coefficients of all second-order sections. scale can be 'none', 'inf' (infinity-norm), or 'two' (2-norm). Using infinity-norm scaling with up ordering minimizes the probability of overflow in the realization. Using 2-norm scaling with down ordering minimizes the peak roundoff noise.
ss	Converts the dfilt to state-space. To see the separate A,B,C,D matrices for the state-space model, use [A,B,C,D]=ss(hd).
stepz	Plots the step response in fvtool
	stepz(hd,n) computes the first n samples of the step response.
	stepz(hd,n,Fs) separates the time samples by T = 1/Fs, where Fs is assumed to be in hertz.
sysobj	Converts the dfilt to a filter System object. See the reference page for a list of supported objects.

dfilt

Method	Description
tf	Converts the dfilt to a transfer function.
zerophase	Plots the zero-phase response in fvtool.
zpk	Converts the dfilt to zeros-pole-gain form.
zplane	Plots a pole-zero plot in fvtool.

Viewing Properties

As with any object, use get to view a dfilt properties. To see a specific property, use

```
get(hd,'property')
```

To see all properties for an object, use

```
get(hd)
```

Note dfilt objects include an arithmetic property. You can change the internal arithmetic of the filter from double- precision to single-precision using: hd.arithmetic = 'single'.

If you have Fixed-Point Toolbox software, you can change the arithmetic property to fixed-point using: hd.arithmetic = 'fixed'

Changing Properties

To set specific properties, use

```
set(hd,'property1',value,'property2',value,...)
```

You must use single quotation marks around the property name. Use single quotation marks around the value argument when the value is a string, such as specifyall or fixed.

Copying an Object

To create a copy of an object, use the copy method.

```
h2 = copy(hd)
```

Note Using the syntax H2 = hd copies only the object handle and does not create a new, independent object.

Converting Between Filter Structures

To change the filter structure of a `dfilt` object `hd`, use

```
hd2 = convert(hd,'structure_string');
```

where `structure_string` is any valid structure name in single quotation marks. If `hd` is a `cascade` or `parallel` structure, each stage is converted to the new structure.

Using Filter States

Two properties control the filter states:

- `states` — Stores the current states of the filter. Before the filter is applied, the states correspond to the initial conditions and after the filter is applied, the states correspond to the final conditions. For `df1`, `df1t`, `df1sos` and `df1tsos` structures, `states` returns a `filtstates` object.

- `PersistentMemory` — Controls whether filter `states` are saved. The default value is `'false'`, which causes the initial conditions to be reset to zero before filtering and turns off the display of `states` information. Setting `PersistentMemory` to `'true'` allows the filter to use your initial conditions or to reuse the final conditions from a previous filtering operation as the initial conditions of the next filtering operation. The `true` setting also displays information about the filter `states`.

Note If you set the `states` and want to use them for filtering, you must set `PersistentMemory` to `'true'` before you use the filter.

---

dfilt

Examples

Create a direct-form I filter, and use a method to see if it is stable.

```
[b,a] = butter(8,0.25);
hd = dfilt.df1(b,a);
isstable(hd)
```

If a dfilt's numerator values do not fit on a single line, a description of the vector is displayed. To see the specific numerator values for this example, use

```
B = get(hd,'numerator');
% or
B1 = hd.numerator;
```

Create an array containing two dfilt objects, apply a method and verify that the method acts on both objects. Use a method to test whether the objects are FIR objects.

```
b = fir1(5,.5);
hd = dfilt.dffir(b);         % Create an FIR filter object
[b,a] = butter(5,.5);        % Create IIR filter
hd(2) = dfilt.df2t(b,a);      % Create DF2T object and place
                              % in the second column of hd.
[h,w] = freqz(hd);
test_fir = isfir(hd)
% hd(1) is FIR and hd(2) is not.
```

Refer to the reference pages for each structure for more examples.

See Also

dfilt | design | fdesign | realizemdl | sos | stepz | dfilt.cascade | dfilt.df1 | dfilt.df1t | dfilt.df2 | dfilt.df2t | dfilt.dfasymfir | dfilt.dffir | dfilt.dffirt | dfilt.dfsymfir | dfilt.latticeallpass | dfilt.latticear | dfilt.latticearma | dfilt.latticemamax | dfilt.latticemamin | dfilt.parallel | dfilt.statespace | filter | freqz | grpdelay | impz | zplane

Purpose   Allpass filter

Syntax   hd = dfilt.allpass(c)

Description hd = dfilt.allpass(c) constructs an allpass filter with the minimum number of multipliers from the elements in vector c. To be valid, c must contain one, two, three, or four real elements. The number of elements in c determines the order of the filter. For example, c with two elements creates a second-order filter and c with four elements creates a fourth-order filter.

The transfer function for the allpass filter is defined by

$$H(z) = \frac{c(n) + c(n-1)z^{-1} + \ldots + z^{-n}}{1 + c(1)z^{-1} + \ldots + c(n)z^{-n}}$$

given the coefficients in c.

To construct a cascade of allpass filter objects, use dfilt.cascadeallpass. For more information about creating cascades of allpass filters, refer to dfilt.cascadeallpass.

Properties The following table provides a list of all the properties associated with an allpass dfilt object.

Property Name	Brief Description
AllpassCoefficients	Contains the coefficients for the allpass filter object
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.

dfilt.allpass

Property Name	Brief Description
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.

Examples This example constructs and displays the information about a second-order allpass filter that uses the minimum number of multipliers.

```
c = [1.5, 0.7];
% Create a second-order dfilt object.
hd = dfilt.allpass(c);
```

See Also dfilt | dfilt.cascadeallpass | dfilt.cascadewdfallpass | dfilt.latticeallpass | mfilt.iirdecim | mfilt.iirinterp

Purpose   Coupled-allpass, lattice filter

Syntax
```
hd = dfilt.calattice(k1,k2,beta)
hd = dfilt.calattice
```

Description   `hd = dfilt.calattice(k1,k2,beta)` returns a discrete-time, coupled-allpass, lattice filter object `hd`, which is two allpass, lattice filter structures coupled together. The lattice coefficients for each structure are vectors `k1` and `k2`. Input argument `beta` is shown in the diagram below.

`hd = dfilt.calattice` returns a default, discrete-time coupled-allpass, lattice filter object, `hd`. The default values are `k1 = k2 = []`, which is the default value for `dfilt.latticeallpass`, and `beta = 1`. This filter passes the input through to the output unchanged.

**calattice
(Coupled–Allpass Lattice)**

dfilt.calattice

Examples   Specify a third-order lattice coupled-allpass filter structure for a `dfilt` filter, `hd` with the following code.

```
k1 = [0.9511 + 1j*0.3088; 0.7511 + 1j*0.1158];
k2 = 0.7502 - 1j*0.1218;
beta = 0.1385 + 1j*0.9904;
hd = dfilt.calattice(k1,k2,beta);
```

The `Allpass1` and `Allpass2` properties store vectors of coefficients.

See Also   dfilt.calatticepc | dfilt | dfilt.latticeallpass | dfilt.latticear | dfilt.latticearma | dfilt.latticemamax | dfilt.latticemamin

Purpose   Coupled-allpass, power-complementary lattice filter

Syntax   hd = dfilt.calatticepc(k1,k2)
                 hd = dfilt.calatticepc

Description hd = dfilt.calatticepc(k1,k2) returns a discrete-time,
                 coupled-allpass, lattice filter object hd, with power-complementary
                 output. This object is two allpass lattice filter structures coupled
                 together to produce complementary output. The lattice coefficients for
                 each structure are vectors, k1 and k2, respectively. beta is shown in the
                 following diagram.

                 hd = dfilt.calatticepc returns a default, discrete-time,
                 coupled-allpass, lattice filter object hd, with power-complementary
                 output. The default values are k1 = k2 = [], which is the default value
                 for the dfilt.latticeallpass. The default for beta = 1. This filter
                 passes the input through to the output unchanged.

dfilt.calatticepc

calatticepc
(Coupled Allpass Lattice, Power Complementary output)

Examples

Specify a third-order lattice coupled-allpass power complementary filter structure for a filter `hd` with the following code. You see from the returned properties that `Allpass1` and `Allpass2` contain vectors of coefficients for the constituent filters.

```
k1 = [0.9511 + 0.3088i; 0.7511 + 0.1158i];
k2 = 0.7502 - 0.1218i;
beta = 0.1385 + 0.9904i;
hd = dfilt.calatticepc(k1,k2,beta);
```

To see the coefficients for `Allpass1`, check the property values.

```
get(hd,'Allpass1')
```

See Also   dfilt.calattice | dfilt | dfilt.latticeallpass |
                dfilt.latticear | dfilt.latticearma | dfilt.latticemamax |
                dfilt.latticemamin

dfilt.cascade

Purpose   Cascade of discrete-time filters

Syntax   Refer to `dfilt.cascade` in Signal Processing Toolbox documentation for more information.

Description `hd = dfilt.cascade(filterobject1,filterobject2,...)` returns a discrete-time filter object `hd` of type `cascade`, which is a serial interconnection of two or more filter objects `filterobject1`, `filterobject2`, and so on. `dfilt.cascade` accepts any combination of `dfilt` objects (discrete time filters) to cascade, as well as Farrow filter objects.

You can use the standard notation to cascade one or more filters:

```
cascade(hd1,hd2,...)
```

where `hd1`, `hd2`, and so on can be mixed types, such as `dfilt` objects and `mfilt` objects.

`hd1`, `hd2`, and so on can be fixed-point filters. All filters in the cascade must be the same arithmetic format — `double`, `single`, or `fixed`. `hd`, the filter object returned, inherits the format of the cascaded filters.

Examples   Cascade a lowpass filter and a highpass filter to produce a bandpass filter.

```
[b1,a1]=butter(8,0.6);     % Lowpass
[b2,a2]=butter(8,0.4,'high');  % Highpass
h1=dfilt.df2t(b1,a1);
h2=dfilt.df2t(b2,a2);
hcas=dfilt.cascade(h1,h2);   % Bandpass with passband 0.4-0.6
% View stage 1 with hcas.Stage(1)
```

See Also   `dfilt` | `dfilt.parallel` | `dfilt.scalar`

Purpose   Cascade of allpass discrete-time filters

Syntax   `hd = dfilt.cascadeallpass(c1,c2,...)`

Description   `hd = dfilt.cascadeallpass(c1,c2,...)` constructs a cascade of allpass filters, each of which uses the minimum number of multipliers, given the filter coefficients provided in `c1`, `c2`, and so on.

Each vector `c` represents one section in the cascade filter. `c` vectors must contain one, two, three, or four elements as the filter coefficients for each section. As a result of the design algorithm, each section is a `dfilt.allpass` structure whose coefficients are given in the matching `c` vector, such as the `c1` vector contains the coefficients for the first stage.

States for each section are shared between sections.

Vectors `c` do not have to be the same length. You can combine various length vectors in the input arguments. For example, you can cascade fourth-order sections with second-order sections, or first-order sections.

For more information about the vectors `ci` and about the transfer function of each section, refer to `dfilt.allpass`.

Generally, you do not construct these allpass cascade filters directly. Instead, they result from the design process for an IIR filter. Refer to the first example in Examples for more about using `dfilt.cascadeallpass` to design an IIR filter.

Properties   In the next table, the row entries are the filter properties and a brief description of each property.

Property Name	Brief Description
AllpassCoefficients	Contains the coefficients for the allpass filter object
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.

Examples

Two examples show how dfilt.cascadeallpass works in very different applications — designing a halfband IIR filter and constructing an allpass cascade of dfilt objects.

First, design the IIR halfband filter using cascaded allpass filters. Each branch of the parallel cascade construction is a cascadeallpass filter object.

```
tw = 100;  % Transition width of filter to be designed, 100 Hz.
ast = 80;  % Stopband attenuation of filter to be designed, 80dB.
fs = 2000; % Sampling frequency of signal to be filtered.
% Store halfband design specs in the specifications object d.
d = fdesign.halfband('tw,ast',tw,ast,fs);
```

Now perform the actual filter design. hd contains two
dfilt.cascadeallpass objects.

```
hd = design(d,'ellip','filterstructure','cascadeallpass');
% Get summary information about one dfilt.cascadeallpass stage.
StageInfo = hd.Stage(1).Stage(1);
```

This second example constructs a dfilt.cascadeallpass filter object
directly given allpass coefficients for the input vectors.

```
section1 = 0.8;
section2 = [1.2,0.7];
section3 = [1.3,0.9];
hd = dfilt.cascadeallpass(section1,section2,section3);
% Get information about the filter
% return informatio in character array
S = info(hd);
```

See Also   dfilt | dfilt.allpass | dfilt.cascadewdfallpass |
                  mfilt.iirdecim | mfilt.iirinterp

dfilt.cascadewdfallpass

Purpose	Cascade allpass WDF filters to construct allpass WDF

Syntax

`hd = dfilt.cascadewdfallpass(c1,c2,...)`

Description

`hd = dfilt.cascadewdfallpass(c1,c2,...)` constructs a cascade of allpass wave digital filters given the allpass coefficients in the vectors `c1`, `c2`, and so on.

Each `c` vector contains the coefficients for one section of the cascaded filter. `C` vectors must have one, two, or four elements (coefficients). Three element vectors are not supported.

When the `c` vector has four elements, the first and third elements of the vector must be 0. Each section of the cascade is an allpass wave digital filter, from `dfilt.wdfallpass`, with the coefficients given by the corresponding c vector. That is, the first section has coefficients from vector `c1`, the second section coefficients come from `c2`, and on until all of the `c` vectors are used.

You can mix the lengths of the `c` vectors. They do not need to be the same length. For example, you can cascade several fourth-order sections (`length(c)` = 4) with first or second-order sections.

Wave digital filters are usually used to create other filters. This toolbox uses them to implement halfband filters, which the first example in Examples demonstrates. They are most often building blocks for filters.

Generally, you do not construct these WDF allpass cascade filters directly. Instead, they result from the design process for an IIR filter. Refer to the first example in Examples for more about using `dfilt.cascadewdfallpass` to design an IIR filter.

For more information about the `c` vectors and the transfer function for the allpass filters, refer to `dfilt.wdfallpass`.

Properties

In the next table, the row entries are the filter properties and a brief description of each property.

Property Name	Brief Description
AllpassCoefficients	Contains the coefficients for the allpass wave digital filter object
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.

Examples   To demonstrate two approaches to using dfilt.cascadewdfallpass to design a filter, these examples show both direct construction and construction as part of another filter.

The first design shown creates an IIR halfband filter that uses lattice wave digital filters. Each branch of the parallel connection in the lattice is an allpass cascade wave digital filter.

```
tw = 100;  % Transition width of filter, 100 Hz.
ast = 80;  % Stopband attenuation of filter, 80 dB.
fs = 2000; % Sampling frequency of signal to filter.
% Store halfband specs.
d = fdesign.halfband('tw,ast',tw,ast,fs);
```

Now perform the actual halfband design process. `hd` contains two `dfilt.cascadewdfallpass` filters.

```
hd = design(d,'ellip','filterstructure','cascadewdfallpass');
% Summary info on dfilt.cascadewdfallpass.
StageSummary = hd.stage(1).stage(2);
```

This example demonstrates direct construction of a `dfilt.cascadewdfallpass` filter with allpass coefficients.

```
section1 = 0.8;
section2 = [1.5,0.7];
section3 = [1.8,0.9];
hd = dfilt.cascadewdfallpass(section1,section2,section3);
```

See Also   dfilt | dfilt.wdfallpass

dfilt.delay

Purpose Delay filter

Syntax
```
Hd = dfilt.delay
Hd = dfilt.delay(latency)
```

Description `Hd = dfilt.delay` returns a discrete-time filter, `Hd`, of type `delay`, which adds a single delay to any signal filtered with `Hd`. The filtered signal has its values shifted by one sample.

`Hd = dfilt.delay(latency)` returns a discrete-time filter, `Hd`, of type `delay`, which adds the number of delay units specified in `latency` to any signal filtered with `Hd`. The filtered signal has its values shifted by the `latency` number of samples. The values that appear before the shifted signal are the filter states.

Examples Create a `delay` filter with a `latency` of 4 and filter a simple signal to view the impact of applying a delay.

```
h = dfilt.delay(4);
Fs = 1000;
t = 0:1/Fs:1;
sig = cos(2*pi*100*t);
y = filter(h,sig);
subplot(211);
stem(sig,'markerfacecolor',[0 0 1]);
axis([0 20 -2 2]);
title('Input Signal');
subplot(212);
stem(y,'markerfacecolor',[0 0 1]);
axis([0 20 -2 2]);
title('Delayed Signal');
```

See Also   dfilt

dfilt.df1

Purpose   Discrete-time, direct-form I filter

Syntax   Refer to `dfilt.df1` in Signal Processing Toolbox documentation.

Description   `hd = dfilt.df1` returns a default discrete-time, direct-form I filter object that uses double-precision arithmetic. By default, the numerator and denominator coefficients b and a are set to 1. With these coefficients the filter passes the input to the output without changes.

Make this filter a fixed-point or single-precision filter by changing the value of the Arithmetic property for the filter hd as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property Arithmetic, refer to "Arithmetic" on page 8-19.

Note a(1), the leading denominator coefficient, cannot be 0. To allow you to change the arithmetic setting to fixed or single, a(1) must be equal to 1.

Fixed-Point Filter Structure   The following figure shows the signal flow for the direct-form I filter implemented by dfilt.df1. To help you see how the filter processes the coefficients, input, output, and states of the filter, as well as numerical operations, the figure includes the locations of the arithmetic and data type format elements within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFormat	`AccumWordLength`	`DenAccumFracLength`	`AccumMode,` `CastBeforeSum`
DenFormat	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale ,` `SignedDenominator`
DenProdFormat	`CoeffWordLength`	`DenProdFracLength`	`ProductMode,` `ProductWordLength`
InputFormat	`InputWordLength`	`InputFracLength`	None
NumAccumFormat	`AccumWordLength`	`NumAccumFracLength`	`AccumMode,` `CastBeforeSum`
NumFormat	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale,` `Signed, Numerator`
NumProdFormat	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength,` `ProductMode`
OutputFormat	`OutputWordLength`	`OutputFracLength`	`OutputMode`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFormat, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFormat refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties   In this table you see the properties associated with df1 implementations of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use `get(hd)` where hd is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Brief Description
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length the filter algorithm uses to interpret the results of product operations involving denominator coefficients. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
Denominator	Stores the denominator coefficients for the IIR filter.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.

Property Name	Brief Description
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputWordLength	Determines the word length used for the output data.

Property Name	Brief Description
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.

Examples   Specify a second-order direct-form I structure for a dfilt object, hd, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1(b,a);
% Convert hd to fixed-point filter
set(hd,'arithmetic','fixed')
```

See Also   dfilt | dfilt.df1t | dfilt.df2 | dfilt.df2t

Purpose   Discrete-time, SOS direct-form I filter

Syntax   Refer to `dfilt.df1sos` in Signal Processing Toolbox documentation.

Description   `hd = dfilt.df1sos(s)` returns a discrete-time, second-order section, direct-form I filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  `set(hd,'arithmetic','single');`

- To change to fixed-point filtering, enter

  `set(hd,'arithmetic','fixed');`

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

`hd = dfilt.df1sos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I filter object `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, and so on.

`hd = dfilt.df1sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. When you do not specify `g`, all gains default to one.

`hd = dfilt.df1sos` returns a default, discrete-time, second-order section, direct-form I filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

dfilt.df1sos

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I filter implemented in second-order sections by `dfilt.df1sos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Similarly consider NumFrmt, which refers to the word and fraction lengths

(CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFrmt	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFrmt	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
DenStateFrmt	DenStateWordLength	DenStateFracLength	CastBeforeSum, States
InputFrmt	InputWordLength	InputFracLength	None
NumAccumFrmt	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFrmt	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
NumStateFrmt	NumStateWordLength	NumStateFracLength	States
OutputFrmt	OutputWordLength	OutputFracLength	OutputMode
ScaleValueFrmt	CoeffWordLength	ScaleValueFracLength	CoeffAutoScale, ScaleValues

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with

product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

Properties
In this table you see the properties associated with SOS implementation of direct-form I dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

get(hd)

where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output

Property Name	Brief Description
	from the accumulator, set `AccumMode` to `SpecifyPrecision`.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.

Property Name	Brief Description
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
DenStateFracLength	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
DenStateWordLength	Specifies the word length used to represent the states associated with denominator coefficients in the filter.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumStateFracLength	Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.

Property Name	Brief Description
NumWordFracLength	Specifies the word length used to interpret the states associated with numerator coefficient operations in the filter.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length applied for the output data.

Property Name	Brief Description
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
ScaleValueFracLength	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable AutoScaleMode by setting it to false.

Property Name	Brief Description
ScaleValues	Scaling for the filter objects in SOS filters.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
SosMatrix	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples   Specify a fixed-point, second-order section, direct-form I dfilt object with the following code:

```
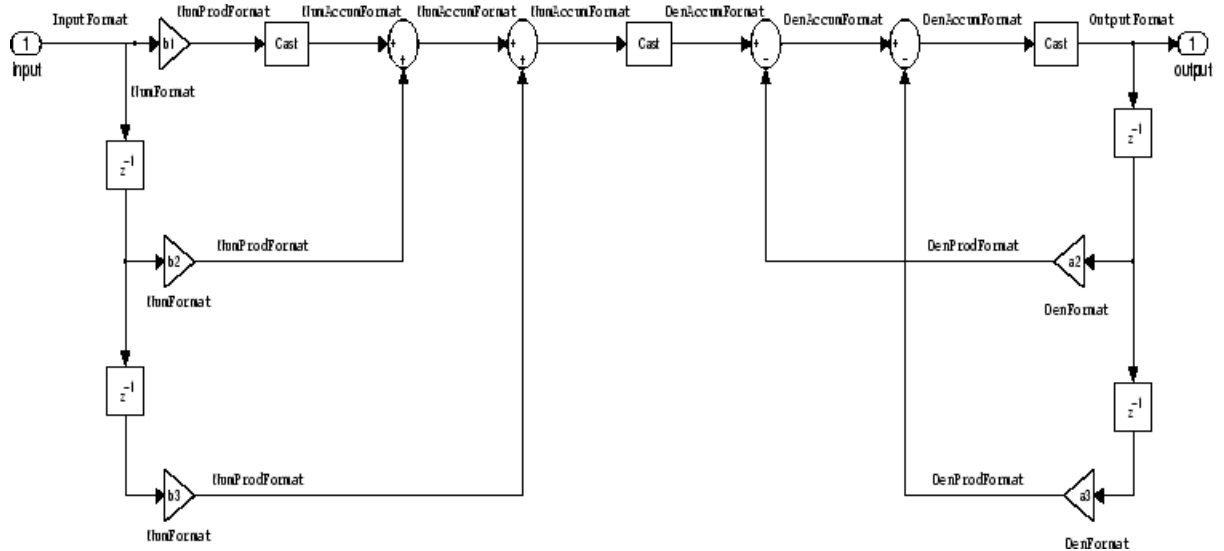b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1sos(b,a);
% Convert to fixed-point filter
hd.arithmetic = 'fixed';
```

See Also   dfilt | dfilt.df2tsos

Purpose Discrete-time, direct-form I transposed filter

Syntax Refer to `dfilt.df1t` in Signal Processing Toolbox documentation.

Description `hd = dfilt.df1t(b,a)` returns a discrete-time, direct-form I transposed filter object hd, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter hd as follows:

- To change to single-precision filtering, enter

  `set(hd,'arithmetic','single');`

- To change to fixed-point filtering, enter

  `set(hd,'arithmetic','fixed');`

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

`hd = dfilt.df1t` returns a default, discrete-time, direct-form I transposed filter object hd, with `b=1` and `a=1`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure The following figure shows the signal flow for the transposed direct-form I filter implemented by `dfilt.df1t`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale`, , `Signed`, `Denominator`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`
DenStateFrmt	`DenStateWordLength`	`DenStateFracLength`	`CastBeforeSum`, `States`
InputFrmt	`InputWordLength`	`InputFracLength`	None
Multiplicandfrmt	`Multiplicand-WordLength`	`Multiplicand-FracLength`	`CastBeforeSum`
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode`, `CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `Signed`, `Numerator`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength`, `ProductMode`
NumStateFrmt	`NumStateWord-Length`	`NumStateFrac-Length`	`States`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From

reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties   In this table you see the properties associated with df1t implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

```
get(hd)
```

where hd is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.

Property Name	Brief Description
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
Denominator	Holds the denominator coefficients for the filter.

Property Name	Brief Description
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
DenStateFracLength	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
MultiplicandFracLength	Sets the fraction length for values (multiplicands) used in multiply operations in the filter.
MultiplicandWordLength	Sets the word length applied to the values input to a multiply operation (the multiplicands).
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.

Property Name	Brief Description
Numerator	Holds the numerator coefficient values for the filter.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
NumStateFracLength	For IIR filters, this defines the binary point location applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data.

Property Name	Brief Description
	• SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

Property Name	Brief Description
	The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples   Specify a second-order direct-form I transposed filter structure for a dfilt object, hd, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1t(b,a);
% Convert filter to single-precision arithmetic
set(hd,'arithmetic','single')
```

See Also   dfilt | dfilt.df1 | dfilt.df2 | dfilt.df2t

Purpose   Discrete-time, SOS direct-form I transposed filter

Syntax   Refer to `dfilt.df1tsos` in Signal Processing Toolbox documentation.

Description   `hd = dfilt.df1tsos(s)` returns a discrete-time, second-order section, direct-form I, transposed filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to .

`hd = dfilt.df1tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form I, transposed filter object `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`hd = dfilt.df1tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`hd = dfilt.df1tsos` returns a default, discrete-time, second-order section, direct-form I, transposed filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

dfilt.df1tsos

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form I transposed filter implemented using second-order sections by `dfilt.df1tsos`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFrmt	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFrmt	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
DenStateFrmt	DenStateWordLength	DenStateFracLength	CastBeforeSum, States
InputFrmt	InputWordLength	InputFracLength	None
MultiplicandFrmt	Multiplicand-WordLength	Multiplicand-FracLength	CastBeforeSum
NumAccumFrmt	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFrmt	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
NumStateFrmt	NumStateWordLength	NumStateFracLength	States
OutputFrmt	OutputWordLength	OutputFracLength	OutputMode
ScaleValueFrmt	CoeffWordLength	ScaleValue-FracLength	CoeffAutoScale, ScaleValues

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties In this table you see the properties associated with SOS implementation of transposed direct-form I `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

`get(hd)`

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.

Property Name	Brief Description
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
DenStateFracLength	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.

Property Name	Brief Description
InputWordLength	Specifies the word length applied to interpret input data.
MultiplicandFracLength	Sets the fraction length for values (multiplicands) used in multiply operations in the filter.
MultiplicandWordLength	Sets the word length applied to the values input to a multiply operation (the multiplicands)
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
NumStateFracLength	For IIR filters, this defines the binary point location applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficient operations in the filter.

Property Name	Brief Description
NumStateWordLength	For IIR filters, this defines the word length applied to the numerator states of the filter. Specifies the word length used to interpret the states associated with numerator coefficient operations in the filter.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.

Property Name	Brief Description
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity.

Property Name	Brief Description
	• round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
	The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
ScaleValueFracLength	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable AutoScaleMode by setting it to false.
ScaleValues	Scaling for the filter objects in SOS filters.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
SosMatrix	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/sectiondatatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples   With the following code, this example specifies a second-order section, direct-form I transposed dfilt object for a filter. Then convert the filter to fixed-point operation.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df1tsos(b,a);
set(hd,'arithmetic','fixed')
```

See Also   dfilt | dfilt.df1sos | dfilt.df2sos | dfilt.df2tsos

Purpose	Discrete-time, direct-form II filter
Syntax	Refer to `dfilt.df2` in Signal Processing Toolbox documentation.
Description	`hd = dfilt.df2(b,a)` returns a discrete-time, direct-form II filter object `hd`, with numerator coefficients `b` and denominator coefficients `a`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

`hd = dfilt.df2` returns a default, discrete-time, direct-form II filter object `hd`, with `b = 1` and `a = 1`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form II filter implemented by `dfilt.df2`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters"frmt" (format). In this use, "frmt" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	AccumWordLength	DenAccumFracLength	AccumMode, CastBeforeSum
DenFrmt	CoeffWordLength	DenFracLength	CoeffAutoScale, Signed, Denominator
DenProdFrmt	CoeffWordLength	DenProdFracLength	ProductMode, ProductWordLength
InputFrmt	InputWordLength	InputFracLength	None
NumAccumFrmt	AccumWordLength	NumAccumFracLength	AccumMode, CastBeforeSum
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
NumProdFrmt	CoeffWordLength	NumProdFracLength	ProductWordLength, ProductMode
OutputFrmt	OutputWordLength	OutputFracLength	OutputMode
StateFrmt	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties ProdWordLength, ProductMode and DenProdFracLength that fully define the denominator format after multiply (or product) operations.

Properties   In this table you see the properties associated with the df2
implementation of dfilt objects.

Note The table lists all the properties that a filter can have. Many of
the properties are dynamic, meaning they exist only in response to the
settings of other properties. You might not see all of the listed properties
all the time. To view all the properties for a filter at any time, use

get(hd)

where hd is a filter.

For further information about the properties of this filter or any dfilt
object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.

Property Name	Brief Description
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
Denominator	Holds the denominator coefficients for IIR filters.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

Property Name	Brief Description
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative

dfilt.df2

Property Name	Brief Description
	numbers, and toward positive infinity for positive numbers.

The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision. |
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples   Specify a second-order direct-form II filter structure for a dfilt object, hd, with the following code:

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df2(b,a);
% Change filter to fixed-point
set(hd,'arithmetic','fixed')
```

See Also   dfilt | dfilt.df1 | dfilt.df1t | dfilt.df2t

Purpose　　　Discrete-time, SOS, direct-form II filter

Syntax　　　Refer to `dfilt.df2sos` in Signal Processing Toolbox documentation.

Description　`hd = dfilt.df2sos(s)` returns a discrete-time, second-order section, direct-form II filter object `hd`, with coefficients given in the `s` matrix.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

`hd = dfilt.df2sos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II object, `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`hd = dfilt.df2sos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`hd = dfilt.df2sos` returns a default, discrete-time, second-order section, direct-form II filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

---

dfilt.df2sos

Fixed-Point Filter Structure

The figure below shows the signal flow for the direct-form II filter implemented with second-order sections by dfilt.df2sos. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The frmt properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths

(`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode,` `CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale,` `Signed, sosMatrix`

DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode,`
			`ProductWordLength,`
			`sosMatrix`

InputFrmt	`InputWordLength`	`InputFracLength`	None
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode,` `CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale,` `Signed, sosMatrix`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength,` `ProductMode`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ScaleValueFrmat	`CoeffWordLength`	`ScaleValue-` `FracLength`	`CoeffAutoScale,` `ScaleValues`
SectionInputFormt	`SectionInput-` `WordLength`	`SectionInput-` `FracLength`	`SectionInput-` `AutoScale`
SectionOutputFrmt	`SectionOutput-` `WordLength`	`SectionOutput-` `FracLength`	`SectionOutput-` `AutoScale`
StateFrmt	`StateWordLength`	`StateFracLength`	`CastBeforeSum,` `States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties   In this table you see the properties associated with second-order section implementation of direct-form II `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

`get(hd)`

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.

Property Name	Brief Description
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.

Property Name	Brief Description
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.

Property Name	Brief Description
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
ScaleValueFracLength	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable AutoScaleMode by setting it to false.

Property Name	Brief Description
ScaleValues	Scaling for the filter objects in SOS filters.
SectionInputAutoScale	Tells the filter whether to set the stage input data format to minimize the occurrence of overflow conditions.
SectionInputFracLength	Lets you set the fraction length for section inputs in SOS filters, if you set `SectionInputAutoScale` to `false`.
SectionInputWordLength	Lets you set the word length for section inputs in SOS filters, if you set `SectionInputAutoScale` to `false`.
SectionOutputAutoScale	Tells the filter whether to set the section output data format to minimize the occurrence of overflow conditions.
SectionOutputFracLength	Lets you set the fraction length for section outputs in SOS filters, if you set `SectionOutputAutoScale` to `off`.
SectionOutputWordLength	Lets you set the word length for section outputs in SOS filters, if you set `SectionOutputAutoScale` to `false`.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
SosMatrix	Holds the filter coefficents as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type `double` to represent the coefficients.

Property Name	Brief Description
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples   Specify a second-order section, direct-form II dfilt object for a Butterworth filter converted to second-order sections, with the following code:

```
[z,p,k] = butter(30,0.5);
[s,g] = zp2sos(z,p,k);
hd = dfilt.df2sos(s,g);
% Convert filter to fixed-point
hd.arithmetic='fixed';
```

See Also   dfilt | dfilt.df1sos | dfilt.df1tsos | dfilt.df2tsos

dfilt.df2t

Purpose	Discrete-time, direct-form II transposed filter
Syntax	Refer to dfilt.df2t in Signal Processing Toolbox documentation.
Description	hd = dfilt.df2t(b,a) returns a discrete-time, direct-form II transposed filter object hd, with numerator coefficients b and denominator coefficients a.

Make this filter a fixed-point or single-precision filter by changing the value of the Arithmetic property for the filter hd as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property Arithmetic, refer to "Arithmetic" on page 8-19.

hd = dfilt.df2t returns a default, discrete-time, direct-form II transposed filter object hd, with b = 1 and a = 1. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator a(1) cannot be 0. To allow you to change the arithmetic setting to fixed or single, a(1) must be equal to 1.

Fixed-Point Filter Structure

The following figure shows the signal flow for the direct-form II transposed filter implemented by dfilt.df2t. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt." In this use, "frmt" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The

format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode, CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale,Signed, Denominator`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode, ProductWordLength`
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode, CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale, Signed, Numerator`
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength, ProductMode`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	None
StateFrmt	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the

properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties   In this table you see the properties associated with `df2t` implementation of `dfilt` objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

`get(hd)`

where `hd` is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.

Property Name	Brief Description
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set AccumMode to SpecifyPrecision.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
Denominator	Holds the denominator coefficients for IIR filters.

Property Name	Brief Description
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set ProductMode to SpecifyPrecision.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set AccumMode to SpecifyPrecision.
Numerator	Holds the numerator coefficient values for the filter.
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OutputFracLength	Determines how the filter interprets the filter output data.
OutputWordLength	Determines the word length used for the output data.

Property Name	Brief Description
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either `saturate` (limit the output to the largest positive or negative representable value) or `wrap` (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (`FullPrecision`), or whether to keep the most significant bit (`KeepMSB`) or least significant bit (`KeepLSB`) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set `ProductMode` to `SpecifyPrecision`.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples   Create a fixed-point filter by specifying a second-order direct-form II transposed filter structure for a dfilt object, and then converting the double-precision arithmetic setting to fixed-point.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df2t(b,a);
% Convert filter to fixed-point
set(hd,'arithmetic','fixed')
```

See Also   dfilt | dfilt.df1 | dfilt.df1t | dfilt.df2

Purpose   Discrete-time, SOS direct-form II transposed filter

Syntax   Refer to `dfilt.df2tsos` in Signal Processing Toolbox documentation.

Description `hd = dfilt.df2tsos(s)` returns a discrete-time, second-order section, direct-form II, transposed filter object `hd`, with coefficients given in the matrix `s`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

• To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

• To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

`hd = dfilt.df2tsos(b1,a1,b2,a2,...)` returns a discrete-time, second-order section, direct-form II, transposed filter object `hd`, with coefficients for the first section given in the `b1` and `a1` vectors, for the second section given in the `b2` and `a2` vectors, etc.

`hd = dfilt.df2tsos(...,g)` includes a gain vector `g`. The elements of `g` are the gains for each section. The maximum length of `g` is the number of sections plus one. If `g` is not specified, all gains default to one.

`hd = dfilt.df2tsos` returns a default, discrete-time, second-order section, direct-form II, transposed filter object, `hd`. This filter passes the input through to the output unchanged.

Note The leading coefficient of the denominator `a(1)` cannot be 0. To allow you to change the arithmetic setting to `fixed` or `single`, `a(1)` must be equal to 1.

Fixed-Point Filter Structure

The figure below shows the signal flow for the second-order section transposed direct-form II filter implemented by dfilt.dftsos. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
DenAccumFrmt	`AccumWordLength`	`DenAccumFracLength`	`AccumMode`, `CastBeforeSum`
DenFrmt	`CoeffWordLength`	`DenFracLength`	`CoeffAutoScale`, `Signed`, `Denominator`
DenProdFrmt	`CoeffWordLength`	`DenProdFracLength`	`ProductMode`, `ProductWordLength`
InputFrmt	`InputWordLength`	`InputFracLength`	None
NumAccumFrmt	`AccumWordLength`	`NumAccumFracLength`	`AccumMode`, `CastBeforeSum`
NumFrmt	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, `SignedNumerator`

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
NumProdFrmt	`CoeffWordLength`	`NumProdFracLength`	`ProductWordLength,` `ProductMode`
OutputFrmt	`OutputWordLength`	`OutputFracLength`	`OutputMode`
ScaleValueFrmt	`CoeffWordLength`	`ScaleValueFracLength`	`CoeffAutoScale,` `ScaleValues`
SectionInputFormt	`SectionInput-` `WordLength`	`SectionInput-` `FracLength`	
SectionOutputFrmt	`SectionOutput-` `WordLength`	`SectionOutput-` `FracLength`	
StateFrmt	`StateWordLength`	`StateFracLength`	`States`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label DenProdFrmt, which always follows a denominator coefficient multiplication element in the signal flow. The label indicates that denominator coefficients leave the multiplication element with the word length and fraction length associated with product operations that include denominator coefficients. From reviewing the table, you see that the DenProdFrmt refers to the properties `ProdWordLength`, `ProductMode` and `DenProdFracLength` that fully define the denominator format after multiply (or product) operations.

Properties   In this table you see the properties associated with second-order section implementation of transposed direct-form II `dfilt` objects.

> **Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use
>
> get(hd)
>
> where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

dfilt.df2tsos

Property Name	Brief Description
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` and `DenFracLength` properties to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Specifies the fraction length used to interpret data in the accumulator used to hold the results of sum operations. You can change the value for this property when you set `AccumMode` to `SpecifyPrecision`.
DenFracLength	Set the fraction length the filter uses to interpret denominator coefficients. `DenFracLength` is always available, but it is read-only until you set `CoeffAutoScale` to `false`.
DenProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value when you set `ProductMode` to `SpecifyPrecision`.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
NumAccumFracLength	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients. You can change the value of this property after you set `AccumMode` to `SpecifyPrecision`.

Property Name	Brief Description
NumFracLength	Sets the fraction length used to interpret the value of numerator coefficients.
NumProdFracLength	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. Available to be changed when you set ProductMode to SpecifyPrecision.
OptimizeScaleValues	When true, the filter skips multiplication-by-one scaling. When false, the filter performs multiplication-by-one scaling.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.

Property Name	Brief Description
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.

Property Name	Brief Description
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
ScaleValueFracLength	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Only available when you disable AutoScaleMode by setting it to false.
ScaleValues	Scaling for the filter objects in SOS filters.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.

Property Name	Brief Description
SosMatrix	Holds the filter coefficients as property values — you use set and get to modify them. Displays the matrix in the format [sections x coefficients/section data type].
SectionInputFracLength	Lets you set the fraction length for section inputs in SOS filters, if you set SectionInputAutoScale to false.
SectionInputWordLength	Lets you set the word length for section inputs in SOS filters, if you set SectionInputAutoScale to false.
SectionOutputFracLength	Lets you set the fraction length for section outputs in SOS filters, if you set SectionOutputAutoScale to off.
SectionOutputWordLength	Lets you set the word length for section outputs in SOS filters, if you set SectionOutputAutoScale to false.
StateAutoScale	Setting autoscaling for filter states to true reduces the possibility of overflows occurring during fixed-point operations. Set to false, StateAutoScale lets the filter select the fraction length to limit the overflow potential.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.
StateWordLength	Sets the word length used to represent the filter states.

Examples   Construct a second-order section Butterworth filter for fixed-point filtering. Start by specifying a Butterworth filter, and then convert the filter to second-order sections, with the following code:

```
[z,p,k] = butter(30,0.5);
```

```
[s,g] = zp2sos(z,p,k);
hd = dfilt.df2tsos(s,g);
% convert filter to fixed-point
hd.arithmetic='fixed';
```

See Also   dfilt | dfilt.df1sos | dfilt.df1tsos | dfilt.df2sos

Purpose	Discrete-time, direct-form antisymmetric FIR filter
Syntax	Refer to `dfilt.dfasymfir` in Signal Processing Toolbox documentation.
Description	`hd = dfilt.dfasymfir(b)` returns a discrete-time, direct-form, antisymmetric FIR filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  `set(hd,'arithmetic','single');`

- To change to fixed-point filtering, enter

  `set(hd,'arithmetic','fixed');`

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

`hd = dfilt.dfasymfir` returns a default, discrete-time, direct-form, antisymmetric FIR filter object `hd`, with b=1. This filter passes the input through to the output unchanged.

Note Only the coefficients in the first half of vector `b` are used because `dfilt.dfasymfir` assumes the coefficients in the second half are antisymmetric to those in the first half. For example, in the figure coefficients, b(4) = -b(3), b(5) = -b(2), and b(6) = -b(1).

Fixed-Point Filter Structure

The following figure shows the signal flow for the odd-order antisymmetric FIR filter implemented by `dfilt.dfasymfir`. The even-order filter uses similar flow. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

dfilt.dfasymfir

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	None
InputFormat	`InputWordLength`	`InputFracLength`	None
NumFormat	`CoeffWordLength`	`NumFracLength`	`CoeffAutoScale`, , `Signed`, `Numerator`
OutputFormat	`OutputWordLength`	`OutputFracLength`	None
ProductFormat	`ProductWordLength`	`ProductFracLength`	None
TapSumFormat	`InputWordLength`	`InputFracLength`	`InputFormat`

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties `ProductFracLength` and `ProductWordLength` that fully define the coefficient format after multiply (or product) operations.

Properties   In this table you see the properties associated with an antisymmetric FIR implementation of `dfilt` objects.

> **Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use
>
> ```
> get(hd)
> ```
>
> where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 8-2.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits [27]	Specifies the fraction length used to interpret data output by the accumulator.
AccumWordLength	Any integer number of bits[33]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to fixed allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

dfilt.dfasymfir

Name	Values	Description
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data. Also controls TapSumFracLength.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data. Also determines TapSumWordLength.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [29]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilerInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [33]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [27]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductWordLength	Any integer number of bits [33]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero.

Name	Values	Description
		• `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	`fi` object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation.

Examples   **Odd Order**

Specify a fifth-order direct-form antisymmetric FIR filter structure for a `dfilt` object, hd, with the following code:

```
b = [-0.008 0.06 -0.44 0.44 -0.06 0.008];
hd = dfilt.dfasymfir(b);
```

Even Order

Specify a fourth-order direct-form antisymmetric FIR filter structure for dfilt object hd, with the following code:

```
b = [-0.01 0.1 0.0 -0.1 0.01];
hd = dfilt.dfasymfir(b);
hd.arithmetic='fixed';
FilterCoefs = get(hd,'numerator');
% or equivalently
FilterCoefs = hd.numerator;
```

See Also   dfilt | dfilt.dffir | dfilt.dffirt | dfilt.dfsymfir

dfilt.dffir

Purpose   Discrete-time, direct-form FIR filter

Syntax   Refer to `dfilt.dffir` in Signal Processing Toolbox documentation.

Description   `hd = dfilt.dffir(b)` returns a discrete-time, direct-form finite impulse response (FIR) filter object `hd`, with numerator coefficients `b`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

`hd = dfilt.dffir` returns a default, discrete-time, direct-form FIR filter object `hd`, with `b=1`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure   The following figure shows the signal flow for the direct-form FIR filter implemented by `dfilt.dffir`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

dfilt.dffir

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFrmt	AccumWordLength	AccumFracLength	None
InputFrmt	InputWordLength	InputFracLength	None
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
OutputFrmt	OutputWordLength	OutputFracLength	None
ProductFrmt	ProductWordLength	ProductFracLength	None

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFrmt, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFrmt refers to the properties ProductFracLength and ProductWordLength that fully define the coefficient format after multiply (or product) operations.

Properties   In this table you see the properties associated with direct-form FIR implementation of dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

get(hd)

where hd is a filter.

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 8-2.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits [30]	Specifies the fraction length used to interpret data output by the accumulator.
AccumWordLength	Any integer number of bits[34]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. `SpecifyPrecision` makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.

Name	Values	Description
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilerInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Name	Values	Description
ProductFracLength	Any positive or negative integer number of bits [30]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductWordLength	Any integer number of bits [32]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient

Name	Values	Description
		and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation.

Examples   Specify a second-order direct-form FIR filter structure for a dfilt object hd, with the following code that constructs the filter in double-precision format and then converts the filter to fixed-point operation:

```
b = [0.05 0.9 0.05];
hd = dfilt.dffir(b);
% Create fixed-point filter
hd.arithmetic='fixed';
% Change FilterInternals property to
% SpecifyPrecision enabling other properites
hd.FilterInternals='SpecifyPrecision';
```

See Also   dfilt | dfilt.dfasymfir | dfilt.dffirt | dfilt.dfsymfir

Purpose   Discrete-time, direct-form FIR transposed filter

Syntax   Refer to dfilt.dffirt in Signal Processing Toolbox documentation.

Description hd = dfilt.dffirt(b) returns a discrete-time, direct-form FIR transposed filter object hd, with numerator coefficients b.

Make this filter a fixed-point or single-precision filter by changing the value of the Arithmetic property for the filter hd as follows:

• To change to single-precision filtering, enter

  set(hd,'arithmetic','single');

• To change to fixed-point filtering, enter

  set(hd,'arithmetic','fixed');

For more information about the property Arithmetic, refer to "Arithmetic" on page 8-19.

hd = dfilt.dffirt returns a default, discrete-time, direct-form FIR transposed filter object hd, with b = 1. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure   The following figure shows the signal flow for the transposed direct-form FIR filter implemented by dfilt.dffirt. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties InputWordLength and InputFracLength (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (CoeffWordLength, NumFracLength) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	AccumWordLength	AccumFracLength	None
InputFormat	InputWordLength	InputFracLength	None
NumFormat	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
OutputFormat	OutputWordLength	OutputFracLength	None
ProductFormat	ProductWordLength	ProductFracLength	None

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength and ProductWordLength that fully define the coefficient format after multiply (or product) operations.

Properties   In this table you see the properties associated with the transposed direct-form FIR implementation of dfilt objects.

> **Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use
>
> ```
> get(hd)
> ```
>
> where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 8-2.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits [30]	Specifies the fraction length used to interpret data output by the accumulator.
AccumWordLength	Any integer number of bits[34]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to fixed allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

Name	Values	Description
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [30]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilerInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [34]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.

dfilt.dffirt

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

Name	Values	Description
		The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	`fi` object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use `fi` objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation.

Examples   Specify a second-order direct-form FIR transposed filter structure for a
`dfilt` object, hd, with the following code:

```
b = [0.05 0.9 0.05];
hd = dfilt.dffirt(b);
set(hd,'arithmetic','fixed')
```

See Also   dfilt | dfilt.dffir | dfilt.dfasymfir | dfilt.dfsymfir

Purpose	Discrete-time, direct-form symmetric FIR filter
Syntax	Refer to dfilt.dfsymfir in Signal Processing Toolbox documentation.
Description	hd = dfilt.dfsymfir(b) returns a discrete-time, direct-form symmetric FIR filter object hd, with numerator coefficients b.

Make this filter a fixed-point or single-precision filter by changing the value of the Arithmetic property for the filter hd as follows:

- To change to single-precision filtering, enter

  set(hd,'arithmetic','single');

- To change to fixed-point filtering, enter

  set(hd,'arithmetic','fixed');

For more information about the property Arithmetic, refer to "Arithmetic" on page 8-19.

hd = dfilt.dfsymfir returns a default, discrete-time, direct-form symmetric FIR filter object hd, with b=1. This filter passes the input through to the output unchanged.

Note Only the coefficients in the first half of vector b are used because dfilt.dfsymfir assumes the coefficients in the second half are symmetric to those in the first half. In the following figure, for example, b(3) = 0, b(4) = b(2) and b(5) = b(1).

Fixed-Point Filter Structure

In the following figure you see the signal flow diagram for the symmetric FIR filter that dfilt.dfsymfir implements.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the letters "frmt" (format). In this use, "frmt" indicates the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFrmt label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFrmt, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFrmt	AccumWordLength	AccumFracLength	None
InputFrmt	InputWordLength	InputFracLength	None
NumFrmt	CoeffWordLength	NumFracLength	CoeffAutoScale, Signed, Numerator
OutputFrmt	OutputWordLength	OutputFracLength	None
ProductFrmt	ProductWordLength	ProductFracLength	None
TapSumFrmt	InputWordLength	InputFracLength	None

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFrmt, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFrmt refers to the properties ProductFracLength and ProductWordLength that fully define the coefficient format after multiply (or product) operations.

Properties   In this table you see the properties associated with the symmetric FIR implementation of dfilt objects.

> **Note** The table lists all the properties that a filter can have. Many of
> the properties are dynamic, meaning they exist only in response to the
> settings of other properties. You might not see all of the listed properties
> all the time. To view all the properties for a filter at any time, use
>
> `get(hd)`
>
> where hd is a filter.

For further information about the properties of this filter or any `dfilt`
object, refer to "Fixed-Point Filter Properties" on page 8-2.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits [27]	Specifies the fraction length used to interpret data output by the accumulator.
AccumWordLength	Any integer number of bits[33]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to `false` enables you to change the `NumFracLength` property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

Name	Values	Description
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [29]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilerInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [33]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [29]	Specifies the fraction length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductWordLength	Any integer number of bits [33]	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Name	Values	Description
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object to match the filter arithmetic setting	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation.

Examples　　**Odd Order**

Specify a fifth-order direct-form symmetric FIR filter structure for a dfilt object hd, with the following code:

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
hd = dfilt.dfsymfir(b);
% Create fixed-point filter
set(hd,'arithmetic','fixed')
% Change FilterInternals property to
% SpecifyPrecision
hd.Filterinternals='SpecifyPrecision';
```

Even Order

Specify a fourth-order, fixed-point, direct-form symmetric FIR filter structure for a dfilt object hd, with the following code:

```
b = [-0.01 0.1 0.8 0.1 -0.01];
hd = dfilt.dfsymfir(b);
set(hd,'arithmetic','fixed');
```

See Also　　dfilt | dfilt.dfasymfir | dfilt.dffir | dfilt.dffirt

dfilt.farrowfd

Purpose   Fractional Delay Farrow filter

Syntax   Hd = dfilt.farrowd(D, COEFFS)

Description   Hd = dfilt.farrowd(D, COEFFS)

Constructs a discrete-time fractional delay Farrow filter with COEFFS coefficients and D delay.

Examples   Farrow filters can be designed with the dfilt.farrowfd filter designer.

```
coeffs = [-1/6 1/2 -1/3 0;1/2 -1 -1/2 1;
-1/2 1/2 1 0;1/6 0 -1/6 0];
Hd = dfilt.farrowfd(0.5, coeffs);
```

Design a cubic fractional delay filter with the Lagrange method.

```
fdelay = .2; % Fractional delay
d = fdesign.fracdelay(fdelay,'N',3);
Hd = design(d, 'lagrange', 'FilterStructure', 'farrowfd');
fvtool(Hd, 'Analysis', 'grpdelay');
```

For more information about fractional delay filter implementations, see the "Fractional Delay Filters Using Farrow Structures" example, farrowdemo.

How To   • dfilt

dfilt.farrowlinearfd

Purpose	Farrow Linear Fractional Delay filter
Syntax	Hd = DFILT.FARROWLINEARFD(D)

Description   Hd = DFILT.FARROWLINEARFD(D)

Constructs a discrete-time linear fractional delay Farrow filter with the delay D.

Examples   Farrow linear fractional delay filter with 1/2 sample delay:

```
 Hd = dfilt.farrowlinearfd(0.5);
n = cos(pi/10*(0:159));
y = filter(Hd,x);
stem(x(1:40));
axis([0 40 -2 2]);
hold on;
stem(y(1:40),'color',[1 0 0],'markerfacecolor',[1 0 0 ]);
legend('Orignal Signal','Filtered Signal','Location','best');
```

For more information about fractional delay filter implementations, see the "Fractional Delay Filters Using Farrow Structures" example, farrowdemo.

How To   • dfilt

Purpose   Discrete-time, overlap-add, FIR filter

Syntax   Hd = dfilt.fftfir(b,len)
                   Hd = dfilt.fftfir(b)
                   Hd = dfilt.fftfir

Description   This object uses the overlap-add method of block FIR filtering, which is
                   very efficient for streaming data.

                   Hd = dfilt.fftfir(b,len) returns a discrete-time, FFT, FIR filter,
                   Hd, with numerator coefficients, b and block length, len. The block
                   length is the number of input points to use for each overlap-add
                   computation.

                   Hd = dfilt.fftfir(b) returns a discrete-time, FFT, FIR filter, Hd,
                   with numerator coefficients, b and block length, len=100.

                   Hd = dfilt.fftfir returns a default, discrete-time, FFT, FIR filter,
                   Hd, with the numerator b=1 and block length, len=100. This filter
                   passes the input through to the output unchanged.

                   Note When you use a dfilt.fftfir object to filter, the input signal
                   length must be an integer multiple of the object's block length, len. The
                   resulting number of FFT points = (filter length + the block length - 1).
                   The filter is most efficient if the number of FFT points is a power of 2.

                   The fftfir uses an overlap-add block processing algorithm, which is
                   represented as follows,

where len is the block length and M is the length of the numerator-1, (length(b)-1), which is also the number of states. The output of each convolution is a block that is longer than the input block by a tail of (length(b)-1) samples. These tails overlap the next block and are added to it. The states reported by dfilt.fftfir are the tails of the final convolution.

Examples   Create an FFT FIR discrete-time filter with coefficients from a 30th order lowpass equiripple design:

```
b = firpm(30,[0 .1 .2 .5]*2,[1 1 0 0]);
Hd = dfilt.fftfir(b);
% To obtain frequency domain coefficients
% used in filtering
Coeffs = fftcoeffs(Hd);
```

See Also   dfilt | dfilt.dffir | dfilt.dfasymfir | dfilt.dffirt | dfilt.dfsymfir

Purpose	Discrete-time, lattice allpass filter
Syntax	Refer to `dfilt.latticeallpass` in Signal Processing Toolbox documentation.
Description	`hd = dfilt.latticeallpass(k)` returns a discrete-time, lattice allpass filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

`hd = dfilt.latticeallpass` returns a default, discrete-time, lattice allpass filter object `hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the allpass lattice filter implemented by `dfilt.latticeallpass`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

dfilt.latticeallpass

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	AccumWordLength	AccumFracLength	AccumMode
InputFormat	InputWordLength	InputFracLength	None
LatticeFormat	CoeffWordLength	LatticeFracLength	CoeffAutoScale
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ProductFormat	ProductWordLength	ProductFracLength	ProductMode
StateFormat	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength, ProductWordLength, and ProductMode that fully define the coefficient format after multiply (or product) operations.

Properties   In this table you see the properties associated with the allpass lattice implementation of dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

get(hd)

where hd is a filter.

---

dfilt.latticeallpass

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumFracLength	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties —`DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Brief Description
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength property value to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Lattice	Any lattice structure coefficients. No default value.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

dfilt.latticeallpass

Property Name	Brief Description
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity.

Property Name	Brief Description
	• nearest - Round toward nearest. Ties round toward positive infinity.
	• round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
	The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples   Specify a third-order lattice allpass filter structure for a dfilt object hd, with the following code:

```
k = [.66 .7 .44];
hd=dfilt.latticeallpass(k);
% convert to fixed-point arithmetic
hd.arithmetic = 'fixed';
```

See Also   dfilt | dfilt.latticear | dfilt.latticearma |
                   dfilt.latticemamax | dfilt.latticemamin

dfilt.latticear

Purpose   Discrete-time, lattice, autoregressive filter

Syntax   Refer to `dfilt.latticear` in Signal Processing Toolbox documentation.

Description `hd = dfilt.latticear(k)` returns a discrete-time, lattice autoregressive filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

`hd = dfilt.latticear` returns a default, discrete-time, lattice autoregressive filter object `hd`, with `k=[ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the autoregressive lattice filter implemented by `dfilt.latticear`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	`AccumWordLength`	`AccumFracLength`	`AccumMode`
InputFormat	`InputWordLength`	`InputFracLength`	None

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
LatticeFormat	CoeffWordLength	LatticeFracLength	CoeffAutoScale
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ProductFormat	ProductWordLength	ProductFracLength	ProductMode
StateFormat	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength, ProductWordLength, and ProductMode that fully define the coefficient format after multiply (or product) operations.

Properties   In this table you see the properties associated with the autoregressive lattice implementation of dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

get(hd)

where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumFracLength	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Brief Description
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Lattice	Any lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

Property Name	Brief Description
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity.

Property Name	Brief Description
	• nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples   Specify a third-order lattice autoregressive filter structure for a dfilt object, hd, with the following code that creates a fixed-point filter.

dfilt.latticear

```
k = [.66 .7 .44];
hd1=dfilt.latticear(k);
hd1.arithmetic='fixed';
specifyall(hd1);
```

See Also   dfilt | dfilt.latticeallpass | dfilt.latticearma |
                dfilt.latticemamax | dfilt.latticemamin

Purpose　Discrete-time, lattice, autoregressive, moving-average filter

Syntax　Refer to `dfilt.latticearma` in Signal Processing Toolbox documentation.

Description　`hd = dfilt.latticearma(k)` returns a discrete-time, lattice moving-average autoregressive filter object `hd`, with lattice coefficients, `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

• To change to single-precision filtering, enter

```
set(hd,'arithmetic','single');
```

• To change to fixed-point filtering, enter

```
set(hd,'arithmetic','fixed');
```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

`hd dfilt.latticearma` returns a default, discrete-time, lattice moving-average, autoregressive filter object `hd`, with `k = [ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure　The following figure shows the signal flow for the autoregressive lattice filter implemented by `dfilt.latticearma`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical operations, the figure includes the locations of the formatting objects within the signal flow.

dfilt.latticearma

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
InputFormat	InputWordLength	InputFracLength	None
LadderAccumFormat	AccumWordLength	LadderAccumFracLength	AccumMode
LadderFormat	CoeffWordLength	LadderFracLength	CoeffAutoScale
LadderProdFormat	ProductWordLength	LadderProdFracLength	ProductMode
LatticeAccumFormat	AccumWordLength	LatticeAccum-FracLength	AccumMode
LatticeFormat	CoeffWordLength	LatticeFracLength	CoeffAutoScale
LatticeProdFormat	ProductWordLength	LatticeProdFracLength	ProductMode
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
StateFormat	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label LatticeProdFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that lattice coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the LatticeProdFormat refers to the properties ProductWordLength, LatticeProdFracLength, and ProductMode that fully define the coefficient format after multiply (or product) operations.

Properties   In this table you see the properties associated with the autoregressive moving-average lattice implementation of dfilt objects.

> **Note** The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use
>
> get(hd)
>
> where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumFracLength	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.

Property Name	Brief Description
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength property to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Ladder	Stores the ladder coefficients for lattice ARMA (dfilt.latticearma) filters.

Property Name	Brief Description
LadderAccumFracLength	Sets the fraction length used to interpret the output from sum operations that include the ladder coefficients. You can change this property value after you set AccumMode to SpecifyPrecision.
LadderFracLength	Determines the precision used to represent the ladder coefficients in ARMA lattice filters.
Lattice	Stores the lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.

Property Name	Brief Description
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set `ProductMode` to `SpecifyPrecision`.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • `ceil` - Round toward positive infinity. • `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • `fix` - Round toward zero. • `floor` - Round toward negative infinity. • `nearest` - Round toward nearest. Ties round toward positive infinity. • `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic.

Property Name	Brief Description
	Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

See Also   dfilt | dfilt.latticeallpass | dfilt.latticear | dfilt.latticemamin | dfilt.latticemamin

dfilt.latticemamax

Purpose	Discrete-time, lattice, moving-average filter with maximum phase
Syntax	Refer to `dfilt.latticemamax` in Signal Processing Toolbox documentation.
Description	`hd = dfilt.latticemamax(k)` returns a discrete-time, lattice, moving-average filter object `hd`, with lattice coefficients `k`.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hd` as follows:

- To change to single-precision filtering, enter

  ```
  set(hd,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hd,'arithmetic','fixed');
  ```

For more information about the property `Arithmetic`, refer to "Arithmetic" on page 8-19.

Note When the `k` coefficients define a maximum phase filter, the resulting filter in this structure is maximum phase. When your coefficients do not define a maximum phase filter, placing them in this structure does not produce a maximum phase filter.

`hd = dfilt.latticemamax` returns a default discrete-time, lattice, moving-average filter object `hd`, with `k = [ ]`. This filter passes the input through to the output unchanged.

Fixed-Point Filter Structure

The following figure shows the signal flow for the maximum phase implementation of a moving-average lattice filter implemented by `dfilt.latticemamax`. To help you see how the filter processes the coefficients, input, and states of the filter, as well as numerical

operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	AccumWordLength	AccumFracLength	AccumMode
InputFormat	InputWordLength	InputFracLength	None
LatticeFormat	CoeffWordLength	LatticeFracLength	CoeffAutoScale
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ProductFormat	ProductWordLength	ProductFracLength	ProductMode
StateFormat	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength, ProductWordLength, and ProductMode that fully define the coefficient format after multiply (or product) operations.

Properties   In this table you see the properties associated with the maximum phase, moving average lattice implementation of dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

get(hd)

where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumFracLength	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (FullPrecision), or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set AccumMode to SpecifyPrecision.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

dfilt.latticemamax

Property Name	Brief Description
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength property to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering—gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Lattice	Any lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

Property Name	Brief Description
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow—they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity.

Property Name	Brief Description
	• nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples Specify a fourth-order lattice, moving-average, maximum phase filter structure for a dfilt object, hd, with the following code:

```
k = [.66 .7 .44 .33];
hd = dfilt.latticemamax(k);
```

See Also   dfilt | dfilt.latticeallpass | dfilt.latticear |
               dfilt.latticearma | dfilt.latticemamin

dfilt.latticemamin

Purpose   Discrete-time, lattice, moving-average filter with minimum phase

Syntax   Refer to `dfilt.latticemamin` in Signal Processing Toolbox
                  documentation.

Description   `hd = dfilt.latticemamin(k)` returns a discrete-time, lattice,
                  moving-average, minimum phase, filter object `hd`, with lattice
                  coefficients `k`.

                  Make this filter a fixed-point or single-precision filter by changing the
                  value of the `Arithmetic` property for the filter `hd` as follows:

                  • To change to single-precision filtering, enter

                    `set(hd,'arithmetic','single');`

                  • To change to fixed-point filtering, enter

                    `set(hd,'arithmetic','fixed');`

                  For more information about the property `Arithmetic`, refer to
                  "Arithmetic" on page 8-19.

                  Note When the `k` coefficients define a minimum phase filter, the
                  resulting filter in this structure is minimum phase. When your
                  coefficients do not define a minimum phase filter, placing them in this
                  structure does not produce a minimum phase filter.

                  `hd = dfilt.latticemamin` returns a default discrete-time, lattice,
                  moving-average, minimum phase, filter object `hd`, with k=[ ]. This filter
                  passes the input through to the output unchanged.

Fixed-Point   The following figure shows the signal flow for the minimum phase
Filter   implementation of a moving-average lattice filter implemented by
Structure   `dfilt.latticemamin`. To help you see how the filter processes the
                  coefficients, input, and states of the filter, as well as numerical

dfilt.latticemamin

operations, the figure includes the locations of the formatting objects within the signal flow.

Notes About the Signal Flow Diagram

To help you understand where and how the filter performs fixed-point arithmetic during filtering, the figure shows various labels associated with data and functional elements in the filter. The following table describes each label in the signal flow and relates the label to the filter properties that are associated with it.

The labels use a common format — a prefix followed by the word "format." In this use, "format" means the word length and fraction length associated with the filter part referred to by the prefix.

For example, the InputFormat label refers to the word length and fraction length used to interpret the data input to the filter. The format properties `InputWordLength` and `InputFracLength` (as shown in the table) store the word length and the fraction length in bits. Or consider NumFormat, which refers to the word and fraction lengths (`CoeffWordLength`, `NumFracLength`) associated with representing filter numerator coefficients.

Signal Flow Label	Corresponding Word Length Property	Corresponding Fraction Length Property	Related Properties
AccumFormat	AccumWordLength	AccumFracLength	AccumMode
InputFormat	InputWordLength	InputFracLength	None
LatticeFormat	CoeffWordLength	LatticeFracLength	CoeffAutoScale
OutputFormat	OutputWordLength	OutputFracLength	OutputMode
ProductFormat	ProductWordLength	ProductFracLength	ProductMode
StateFormat	StateWordLength	StateFracLength	States

Most important is the label position in the diagram, which identifies where the format applies.

As one example, look at the label ProductFormat, which always follows a coefficient multiplication element in the signal flow. The label indicates that coefficients leave the multiplication element with the word length and fraction length associated with product operations that include coefficients. From reviewing the table, you see that the ProductFormat refers to the properties ProductFracLength, ProductWordLength, and ProductMode that fully define the coefficient format after multiply (or product) operations.

Properties   In this table you see the properties associated with the minimum phase, moving average lattice implementation of dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

get(hd)

where hd is a filter.

---

dfilt.latticemamin

For further information about the properties of this filter or any `dfilt` object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
AccumFracLength	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — `DenAccumFracLength` and `NumAccumFracLength` — that let you set the precision for numerator and denominator operations separately.
AccumMode	Determines how the accumulator outputs stored values. Choose from full precision (`FullPrecision`), or whether to keep the most significant bits (`KeepMSB`) or least significant bits (`KeepLSB`) when output results need shorter word length than the accumulator supports. To let you set the word length and the precision (the fraction length) used by the output from the accumulator, set `AccumMode` to `SpecifyPrecision`.
AccumWordLength	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	Defines the arithmetic the filter uses. Gives you the options `double`, `single`, and `fixed`. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Brief Description
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the LatticeFracLength property to specify the precision used.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Specifies the word length applied to interpret input data.
Lattice	Any lattice structure coefficients.
LatticeFracLength	Sets the fraction length applied to the lattice coefficients.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.

Property Name	Brief Description
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
ProductFracLength	For the output from a product operation, this sets the fraction length used to interpret the data. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.

Property Name	Brief Description
ProductMode	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.
ProductWordLength	Specifies the word length to use for multiplication operation results. This property becomes writable (you can change the value) when you set ProductMode to SpecifyPrecision.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity.

dfilt.latticemamin

Property Name	Brief Description
	• nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
StateFracLength	When you set StateAutoScale to false, you enable the StateFracLength property that lets you set the fraction length applied to interpret the filter states.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.
StateWordLength	Sets the word length used to represent the filter states.

Examples   Specify a third-order lattice, moving-average, minimum phase, filter structure for a dfilt object, hd, with the following code:

```
k = [.66 .7 .44];
hd = dfilt.latticemamin(k);
```

See Also   dfilt | dfilt.latticeallpass | dfilt.latticear |
                 dfilt.latticearma | dfilt.latticemamax

dfilt.parallel

Purpose   Discrete-time, parallel structure filter

Syntax   Refer to `dfilt.parallel` in Signal Processing Toolbox documentation.

Description   `hd = dfilt.parallel(hd1,hd2,...)` returns a discrete-time filter object `hd`, which is a structure of two or more `dfilt` filter objects, `hd1`, `hd2`, and so on arranged in parallel.

You can also use the standard notation to combine filters into a parallel structure.

`parallel(hd1,hd2,...)`

In this syntax, `hd1`, `hd2`, and so on can be a mix of `dfilt` objects, `mfilt` objects, and `adaptfilt` objects.

`hd1`, `hd2`, and so on can be fixed-point filters. All filters in the parallel structure must be the same arithmetic format — double, single, or fixed. `hd`, the filter returned, inherits the format of the individual filters.

See Also   `dfilt` | `dfilt.cascade` | `parallel` | `dfilt.cascade` | `dfilt.parallel`

Purpose   Discrete-time, scalar filter

Syntax   Refer to dfilt.scalar in Signal Processing Toolbox documentation.

Description dfilt.scalar(g) returns a discrete-time, scalar filter object with gain g, where g is a scalar.

Make this filter a fixed-point or single-precision filter by changing the value of the Arithmetic property for the filter hd as follows:

• To change to single-precision filtering, enter

  set(hd,'arithmetic','single');

• To change to fixed-point filtering, enter

  set(hd,'arithmetic','fixed');

For more information about the property Arithmetic, refer to "Arithmetic" on page 8-19.

dfilt.scalar returns a default, discrete-time scalar gain filter object hd, with gain 1.

Properties In this table you see the properties associated with the scalar implementation of dfilt objects.

Note The table lists all the properties that a filter can have. Many of the properties are dynamic, meaning they exist only in response to the settings of other properties. You might not see all of the listed properties all the time. To view all the properties for a filter at any time, use

get(hd)

where hd is a filter.

For further information about the properties of this filter or any dfilt object, refer to "Fixed-Point Filter Properties" on page 8-2.

Property Name	Brief Description
Arithmetic	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.
CoeffAutoScale	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the CoeffFracLength property to specify the precision used.
CoeffFracLength	Set the fraction length the filter uses to interpret coefficients. CoeffFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
CoeffWordLength	Specifies the word length to apply to filter coefficients.
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
Gain	Returns the gain for the scalar filter. Scalar filters do not alter the input data except by adding gain.
InputFracLength	Specifies the fraction length the filter uses to interpret input data.

Property Name	Brief Description
InputWordLength	Specifies the word length applied to interpret input data.
OutputFracLength	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set OutputMode to SpecifyPrecision.
OutputMode	Sets the mode the filter uses to scale the filtered data for output. You have the following choices: • AvoidOverflow — directs the filter to set the output data word length and fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data word length and fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the word and fraction lengths used by the output data from filtering.
OutputWordLength	Determines the word length used for the output data.
OverflowMode	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic). The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. False is the default setting.
RoundMode	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Property Name	Brief Description
Signed	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to filtstates in Signal Processing Toolbox documentation or in the Help system.

Examples   Create a direct-form I filter object hd_filt and a scalar object with a gain of 3 hd_gain and cascade them together.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd_filt = dfilt.df1(b,a);
hd_gain = dfilt.scalar(3);
hd_cascade=cascade(hd_gain,hd_filt);
fvtool_handle = fvtool(hd_filt,hd_gain,hd_cascade);
legend(fvtool_handle,'IIR Filter (direct form I)',...
'Gain','Cascaded Filter');
```

To view the stages of the cascaded filter, use

hd.Stage(1)

and

hd.Stage(2)

See Also   dfilt | dfilt.cascade

Purpose   Wave digital allpass filter

Syntax   hd = dfilt.wdfallpass(c)

Description   hd = dfilt.wdfallpass(c) constructs an allpass wave digital filter structure given the allpass coefficients in vector c.

Vector c must have, one, two, or four elements (filter coefficients). Filters with three coefficients are not supported. When you use c with four coefficients, the first and third coefficients must be 0.

Given the coefficients in c, the transfer function for the wave digital allpass filter is defined by

$$H(z) = \frac{c(n) + c(n-1)z^{-1} + \ldots + z^{-n}}{1 + c(1)z^{-1} + \ldots + c(n)z^{-n}}$$

Internally, the allpass coefficients are converted to wave digital filters for filtering. Note that dfilt.wdfallpass allows only stable filters. Also note that the leading coefficient in the denominator, a 1, does not need to be included in vector c.

Use the constructor dfilt.cascadewdfallpass to cascade wdfallpass filters.

To compare these filters to other similar filters, dfilt.wdfallpass and dfilt.cascadewdfallpass filters have the same number of multipliers as the non-wave digital filters dfilt.allpass and dfilt.cascadeallpass. However, the wave digital filters use fewer states and they may require more adders in the filter structure.

Wave digital filters are usually used to create other filters. This toolbox uses them to implement halfband filters, which the first example in Examples demonstrates. They are most often building blocks for filters.

Properties   In the next table, the row entries are the filter properties and a brief description of each property.

dfilt.wdfallpass

Property Name	Brief Description
AllpassCoefficients	Contains the coefficients for the allpass wave digital filter object
FilterStructure	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. `False` is the default setting.
States	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. They also provide linkage between the sections of a multisection filter, such as a cascade filter. For details, refer to `filtstates` in Signal Processing Toolbox documentation or in the Help system.

Filter Structure

When you change the order of the wave digital filters in the cascade, the filter structure changes as well.

As shown in this example, `realizemdl` lets you see the filter structure used for your filter, if you have Simulink installed.

```
section11=0.8;
section12=[1.5,0.7];
section13=[1.8,0.9];
hd1=dfilt.cascadewdfallpass(section11,section12,section13);


section21=[0.8,0.4];
section22=[0,1.5,0,0.7];
```

```
section23=[0,1.8,0,0.9];
hd2=dfilt.cascadewdfallpass(section21,section22,section23);
% If you have Simulink
realizemdl(hd2)
```

hd1 has this filter structure with three sections.

The filter structure for `hd2` is somewhat different, with the different orders and interconnections between the three sections.

Examples   Construct a second-order wave digital allpass filter with two coefficients. Note that to use realizemdl, you must have Simulink.

```
c = [1.5,0.7];
hd = dfilt.wdfallpass(c);
```

With Simulink installed, realizemdl returns this structure for hd.

See Also   dfilt | dfilt.allpass | dfilt.latticeallpass | dfilt.cascadewdfallpass | dfilt.cascadeallpass | mfilt.iirdecim | mfilt.iirinterp

disp

Purpose	Filter properties and values

Syntax	`disp(hd)`
	`disp(ha)`
	`disp(hm)`

Description	Similar to omitting the closing semicolon from an expression on the command line, except that `disp` does not display the variable name. `disp` lists the property names and property values for any filter object, such as a `dfilt` object or `adaptfilt` object. The following examples illustrate the default display for an adaptive filter `ha` and a multirate filter `hm`.

```
ha=adaptfilt.rls;
fprintf('Displaying properties for default RLS adaptive filter:\n');
fprintf('\n');
disp(ha)
pause(2);
hm=mfilt.cicdecim(6);
fprintf('Displaying properties for cascaded integrator-comb decimator\n')
fprintf('with a decimation factor of 6:\n');
fprintf('\n');
disp(hm)
```

See Also	`set`

Purpose   Cast fixed-point filter to use double-precision arithmetic

Syntax   hd = double(h)

Description hd = double(h) returns a new filter hd that has the same structure
and coefficients as h, but whose arithmetic property is set to double
to use double-precision arithmetic for filtering. double(h) is not the
same as the reffilter(h) function:

- hd, the filter returned by double has the quantized coefficients of h
  represented in double-precision floating-point format

- The reference filter returned by reffilter has double-precision,
  floating-point coefficients that have not been quantized.

You might find double(h) useful to isolate the effects of quantizing the
coefficients of a filter by using double to create a filter hd that operates
in double-precision but uses the quantized filter coefficients.

Examples   Use the same filter, once with fixed-point arithmetic and once with
floating-point, to compare fixed-point filtering with double-precision
floating-point filtering.

```
h = dfilt.dffir(firgr(27,[0 .4 .6 1],...
[1 1 0 0]));                    % Lowpass filter.
% Set h to use fixed-point arithmetic to filter.
% Quantize the coeffs.
h.arithmetic = 'fixed';
% Cast h to double-precision
hd = double(h);
% Set up an input signal.
n = 0:99; x = sin(0.7*pi*n(:));
y = filter(h,x);                % Fixed-point output.
yd = filter(hd,x);              % Floating-point output.
FixedFloatNormDiff=norm(yd-double(y),inf);
```

See Also   reffilter

dsp.util.getLogsArray

Purpose	Return logged signal as MATLAB array

Syntax

```
Array = dsp.util.getLogsArray(LogObject,Format2D,
    'SignalPath',PATH)
Array = dsp.util.getLogsArray(LogObject,Format2D,
    'SignalName',NAME)
```

Description

Array =
dsp.util.getLogsArray(LogObject,Format2D,'SignalPath',PATH)
returns a MATLAB array that contains a signal in `LogObject`. You
must specify the `PATH` to the signal in `LogObject` using the
`Name,Value` pair argument.

Array =
dsp.util.getLogsArray(LogObject,Format2D,'SignalName',NAME)
returns a MATLAB array that contains a signal in `LogObject`. You
must specify the `NAME` of the signal in `LogObject` using the
`Name,Value` pair argument.

Input Arguments

LogObject

Specify the name of the object that contains your logged signals. Valid
classes for `LogObject` depend on the syntax you use:

- When you specify PATH as a `dsp.util.SignalPath` object,
  `LogObject` can be either a `Simulink.SimulationData.Dataset` or
  `Simulink.SimulationData.Signal` object.

- When you specify PATH as the full path to a block in a Simulink
  model, `LogObject` must be a `Simulink.SimulationData.Dataset`
  object.

- When you specify the NAME of a signal in
  `LogObject`, `LogObject` can be an object of class
  `timeseries`, `Simulink.SimulationData.Dataset`, or
  `Simulink.SimulationData.Signal`.

Format2D

Specify a logical value to determine whether the function formats 3-D logged signals as a 2-D or 3-D MATLAB array. When you set this property to `true`, the function uses the following formula to format the 3-D logged signal into a 2-D MATLAB array:

```
dim = size(signal);
ntimes = dim(1)*dim(3);
Array = reshape(permute(signal,[1 3 2]),[ntimes dim(2)]);
```

When you set this property to `false`, the function returns the logged signal without any reformatting.

PATH

Specify the path to the logged signal in `LogObject`. You can specify the path using a `dsp.util.SignalPath` object, or you can provide the full path to a block in your Simulink model. To get the full path to a block in your Simulink model, use the `gcb` command.

NAME

Specify the name of the signal in `LogObject`.

Output Arguments

Array

The output `Array` is a MATLAB array that contains the specified logged signal. When the input is a 3-D logged signal, the dimensions of `Array` depend on the value you specify for `Format2D`:

- When `Format2D` is `true`, `Array` is a 2-D MATLAB array.

- When `Format2D` is `false`, `Array` is a 3-D MATLAB array.

When the input is not a 3-D signal, the dimensions of the output `Array` match those of the input.

dsp.util.getLogsArray

Examples

> **Note** To run the following examples, you must first load
> ex_logsout.mat which contains a `Simulink.SimulationData.Dataset`
> object. Alternatively, you can open and simulate the ex_log_utils
> Simulink model. Doing so will log signals and generate the necessary
> `ex_logsout` object.

Extract a unique signal named Signal3x4 from ex_logsout.

```
dsp.util.getLogsArray(ex_logsout, true, 'SignalName','Signal3x4')
```

Extract a unique signal named Signal3x4 from ex_logsout as a 3-D array.

```
dsp.util.getLogsArray(ex_logsout, false, 'SignalName','Signal3x4')
```

Find and extract a specific signal from multiple signals that have the same name.

Because ex_logsout contains multiple signals named Signal2x4, you
must use the `dsp.util.getSignalPath` function to find the paths to
each of those signals.

```
paths = dsp.util.getSignalPath(ex_logsout, 'Signal2x4')
% paths is a 2x1 array of dsp.util.SignalPath objects. Next, examine
% the BlockPath property of each paths object.
paths.BlockPath
% Find the signal path that corresponds to the logged signal you are
% interested in. For example paths(2). You can then use the
% dsp.util.getLogsArray function and provide the 'SignalPath' name-value
% pair argument.
dsp.util.getLogsArray(ex_logsout, true, 'SignalPath', paths(2))
```

Find and extract a signal from a bus.

Use the `dsp.util.getSignalPath` function to get paths to all the
signals in the bus named Bus1.

```
buspaths = dsp.util.getSignalPath(ex_logsout, 'Bus1')
% buspaths is a 2x1 array of dsp.util.SignalPath objects. Examine the
% BusElement property of each buspaths object.
buspaths.BusElement
% Select a signal path. For example buspaths(1). This is the path to t
% signal named 'Signal3x4' in bus 'Bus' that is contained in bus 'Bus
% Now that you have the path to the signal, call dsp.util.getLogsArray
% using the 'SignalPath' name-value pair argument.
dsp.util.getLogsArray(ex_logsout, true, 'SignalPath', buspaths(1))
```

See Also   dsp.util.getSignalPath | dsp.util.SignalPath
| Simulink.SimulationData.BlockPath
| Simulink.SimulationData.Dataset |
Simulink.SimulationData.Signal | timeseries

How To   • "Export Signal Data Using Signal Logging"

• "Configure a Signal for Signal Logging"

• "Specify the Signal Logging Data Format"

dsp.util.getSignalPath

Purpose	Paths to logged signals
Syntax	`Path = dsp.util.getSignalPath(LogObject, SignalName)`
Description	`Path = dsp.util.getSignalPath(LogObject, SignalName)` returns all paths to signals in `LogObject` with name `SignalName`. The output `Path` is a `dsp.util.SignalPath` object or an array of `dsp.util.SignalPath` objects.
Tips	• To return the path to an unnamed signal in `LogObject`, set `SignalName` to the empty string (`''`).

Input Arguments

LogObject

Specify the name of the object that contains your logged signals. `LogObject` must be a `Simulink.SimulationData.Dataset` or `Simulink.SimulationData.Signal` object.

SignalName

Specify the name of a logged signal in `LogObject`.

Output Arguments

Path

The output `Path` contains the path to all signals named `SignalName` in `LogObject`.

- If `LogObject` contains a unique signal with name `SignalName`, the function returns a single `dsp.util.SignalPath` object.

- If `LogObject` contains more than one signal with the specified name, the function returns an array of `dsp.util.SignalPath` objects.

Examples

Note To run the following examples, you must first load
ex_logsout.mat which contains a Simulink.SimulationData.Dataset
object. Alternatively, you can open and simulate the ex_log_utils
Simulink model. Doing so will log signals and generate the necessary
ex_logsout object.

Find and extract a specific signal from multiple signals that have the same name.

Because ex_logsout contains multiple signals named Signal2x4, you
must use the dsp.util.getSignalPath function to find the paths to
each of those signals.

```
paths = dsp.util.getSignalPath(ex_logsout, 'Signal2x4')
% paths is a 2x1 array of dsp.util.SignalPath objects. Next, examine
% the BlockPath property of each paths object.
paths.BlockPath
% Find the signal path that corresponds to the logged signal you are
% interested in. For example paths(2). You can then use the
% dsp.util.getLogsArray function and provide the 'SignalPath' name-val
% pair argument.
dsp.util.getLogsArray(ex_logsout, true, 'SignalPath', paths(2))
```

Find and extract a signal from a bus.

Use the dsp.util.getSignalPath function to get paths to all the
signals in the bus named Bus1.

```
buspaths = dsp.util.getSignalPath(ex_logsout, 'Bus1')
% buspaths is a 2x1 array of dsp.util.SignalPath objects. Examine the
% BusElement property of each buspaths object.
buspaths.BusElement
% Select a signal path. For example buspaths(1). This is the path to t
% signal named 'Signal3x4' in bus 'Bus' that is contained in bus 'Bus'
% Now that you have the path to the signal, call dsp.util.getLogsArray
% using the 'SignalPath' name-value pair argument.
dsp.util.getLogsArray(ex_logsout, true, 'SignalPath', buspaths(1))
```

dsp.util.getSignalPath

See Also
dsp.util.getLogsArray | dsp.util.SignalPath
| Simulink.SimulationData.Dataset
| Simulink.SimulationData.Signal |
Simulink.SimulationData.updateDatasetFormatLogging

How To
- "Export Signal Data Using Signal Logging"
- "Configure a Signal for Signal Logging"
- "Specify the Signal Logging Data Format"

Purpose   Properties of paths to signals

Description   `dsp.util.SignalPath` objects contain path information for
                  signals in `Simulink.SimulationData.Dataset` objects. You get
                  `Simulink.SimulationData.Dataset` objects when you use `Dataset`
                  logging to log signals from a Simulink model.

Construction `dsp.util.SignalPath` objects are returned by the
                  `dsp.util.getSignalPath` function and can be used as input
                  to the `dsp.util.getLogsArray` function.

Properties   **SignalName**

                  Contains the name of the signal output by the block at the
                  specified `BlockPath`.

                  BlockPath

                  Provides the `Simulink.SimulationData.BlockPath` to the block
                  in the Simulink model from which the signal originates.

                  PortIndex

                  Provides the output port index of the block from which the logged
                  signal `SignalName` originates.

                  BusElement

                  Provides a string description of a signal in a logged bus. When
                  `SignalPath` is not a logged bus, this property will be an empty
                  string.

                  If the `SignalPath` object is a logged bus signal, the `BusElement`
                  string will be formatted as follows:

                  • When the bus contains a nonbus signal, `BusElement` is the
                    name of that signal.

                  • When the bus contains a nested bus that contains a nonbus
                    signal, `BusElement` will be a dot-separated string consisting of

the name of the nested bus followed by the name of the non-bus signal. For example: `nestbus.signal1`

- When the bus contains nested busses within nested busses to any depth, `BusElement` will be a dot-separated string. This string contains each of the nested bus names, and ends with the nonbus signal name. For example: `outernestedbus.innernestedbus.signal1`

See Also   dsp.util.getLogsArray | dsp.util.getSignalPath
| Simulink.SimulationData.BlockPath |
Simulink.SimulationData.Dataset

Purpose	Identify whether blocks in model are current, deprecated, or obsolete
Syntax	`dsp_links` `dsp_links('modelname')`
Description	`dsp_links` returns a structure with three elements that identify whether the DSP System Toolbox blocks in the current model are current, deprecated, or obsolete. Each element is one of the three block categories and contains a cell array of strings. Each string is the name of a library block in the current model. `dsp_links('modelname')` returns the three-element structure for the specified model.

Definitions

Obsolete Blocks

Obsolete blocks are blocks that the toolbox no longer supports. In some cases, these blocks no longer function properly.

Deprecated Blocks

Deprecated blocks are blocks that the toolbox still supports but are likely to become obsolete in a future release. Refer to the block reference page for suggested replacements.

Current Blocks

Current blocks are blocks that the toolbox supports and that represent the latest block functionality.

Examples

Display block support information for the specified model, and then find the name of the first current block:

```
sys = 'dspcochlear';
load_system(sys)     % Load the dspcochlear model
links = dsp_links(sys) % Run dsp_links on the model
links.current{1}     % Find the name of the first current block
```

See Also   `liblinks`

dsplib

Purpose   Open top-level DSP System Toolbox library

Syntax   `dsplib`

Description   `dsplib` opens the top-level DSP System Toolbox block library model.

Examples   View and gain access to the DSP System Toolbox blocks:

`dsplib`

Alternatives   To view and gain access to the DSP System Toolbox blocks using the Simulink library browser:

- Type `simulink` at the MATLAB command line, and then expand the DSP System Toolbox node in the library browser.

- Click the Simulink icon from the MATLAB Toolstrip.

Purpose   Configure Simulink environment for signal processing systems

Syntax   dspstartup

Description   dspstartup configures Simulink environment parameters with settings
                   appropriate for a typical signal processing project. You can use the
                   dspstartup function in the following ways:

- At the MATLAB command line. Doing so configures the Simulink
  environment in your current session for signal processing projects.

- By adding a call to the dspstartup function from your startup.m file.
  When you do so, MATLAB configures your Simulink environment for
  typical signal processing projects each time you launch MATLAB.

When the function successfully configures your Simulink environment,
MATLAB displays the following message in the command window.

```
Changed default Simulink settings for signal processing
systems (dspstartup.m).
```

The dspstartup.m file executes the following commands. See
set_param in the Simulink documentation.

```
set_param(0, ...
          'SingleTaskRateTransMsg','error', ...
          'multiTaskRateTransMsg', 'error', ...
          'Solver',                'fixedstepdiscrete', ...
          'SolverMode',            'SingleTasking', ...
          'StartTime',             '0.0', ...
          'StopTime',              'inf', ...
          'FixedStep',             'auto', ...
          'SaveTime',              'off', ...
          'SaveOutput',            'off', ...
          'AlgebraicLoopMsg',      'error', ...
          'SignalLogging',         'off');
```

dspstartup

Examples
Add a call to the dspstartup function from your startup.m file:

1 To find out if there is a startup.m file on your MATLAB path, run the following code at the MATLAB command line:

```
which startup.m
```

2 If MATLAB returns 'startup.m' not found., see "Startup Options" in the MATLAB documentation to learn more about the startup.m file.

If MATLAB returns a path to your startup.m file, open that file for editing.

```
edit startup.m
```

3 Add a call to the dspstartup function. Your startup.m file now resembles the following code sample:

```
%STARTUP    Startup file
%    This file is executed when MATLAB starts up, if it exists
%    anywhere on the path.  In this case, the startup.m file
%    runs the dspstartup.m file to configure the Simulink
%    environment with settings appropriate for typical
%    signal processing projects.

dspstartup;
```

See Also
startup

How To
• "Configure the Simulink Environment for Signal Processing Models"

Purpose   Elliptic filter using specification object

Syntax   hd = design(d,'ellip')
               hd = design(d,'ellip',designoption,value,designoption,...
               value,...)

Description   hd = design(d,'ellip') designs an elliptical IIR digital filter using
               the specifications supplied in the object h.

               hd = design(d,'ellip',designoption,value,designoption,...
               value,...) returns an elliptical or Cauer FIR filter where you specify
               design options as input arguments.

               To determine the available design options, use designopts with the
               specification object and the design method as input arguments as
               shown.

               designopts(d,'method')

               For complete help about using ellip, refer to the command line help
               system. For example, to get specific information about using ellip with
               d, the specification object, enter the following at the MATLAB prompt.

               help(d,'ellip')

Examples   These examples demonstrate how to use ellip to design filters based
               on filter specification objects.

               Example 1

               Construct the default bandpass filter specification object and design
               an elliptic filter.

               d = fdesign.bandpass;
               hd = design(d,'ellip','matchexactly','both');

Example 2

Construct a lowpass object with order, passband-edge frequency, stopband-edge frequency, and passband ripple specifications, and then design an elliptic filter.

```
d = fdesign.lowpass('n,fp,fst,ap',6,20,25,.8,80);
design(d,'ellip'); % Starts FVtool to display the filter.
```


Example 3

Construct a lowpass object with filter order, passband edge frequency, passband ripple, and stopband attenuation specifications, and then design an elliptic filter.

```
d = fdesign.lowpass('n,fp,ap,ast',6,20,.8,60,80);
design(d,'ellip'); % Starts FVTool to display the filter.
```

See Also   butter | cheby1 | cheby2

euclidfactors

Purpose   Euclid factors for multirate filter

Syntax   `[lo,mo] = euclidfactors(hm)`

Description   `[lo,mo] = euclidfactors(hm)` returns integer factors `lo` and `mo` such that (`lo*L`)-(`mo*M`) = -1. L and M are relatively prime and represent the interpolation and decimation factors of the multirate filter `hm`.

`euclidfactors` works with the multirate filters `mfilt.firfracdecim` and `mfilt.firfracinterp`. You cannot return `lo` and `mo` for decimators or interpolators.

Examples   Use an FIR fractional decimator, with L = 5 and M = 7, to show what `euclidfactors` does.

Indeed, (`lo*L`)-(`mo*M`) = (4*5)-(3*7) = -1.

See Also   `polyphase` | `nstates`

Purpose   Equiripple single-rate or multirate FIR filter from specification object

Syntax   ```
hd = design(d,'equiripple')
hd = design(d,'equiripple',designoption,value,designoption,
...value,...)
```
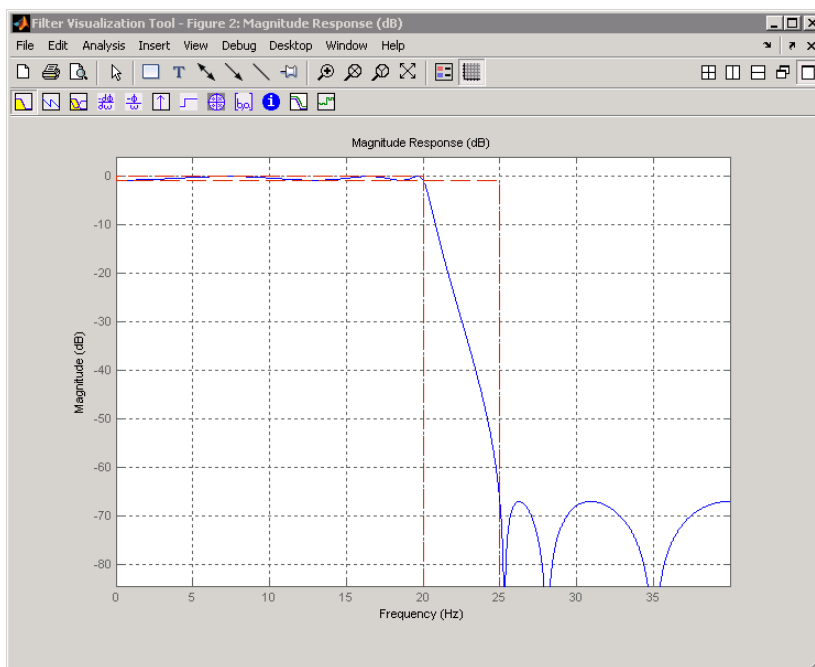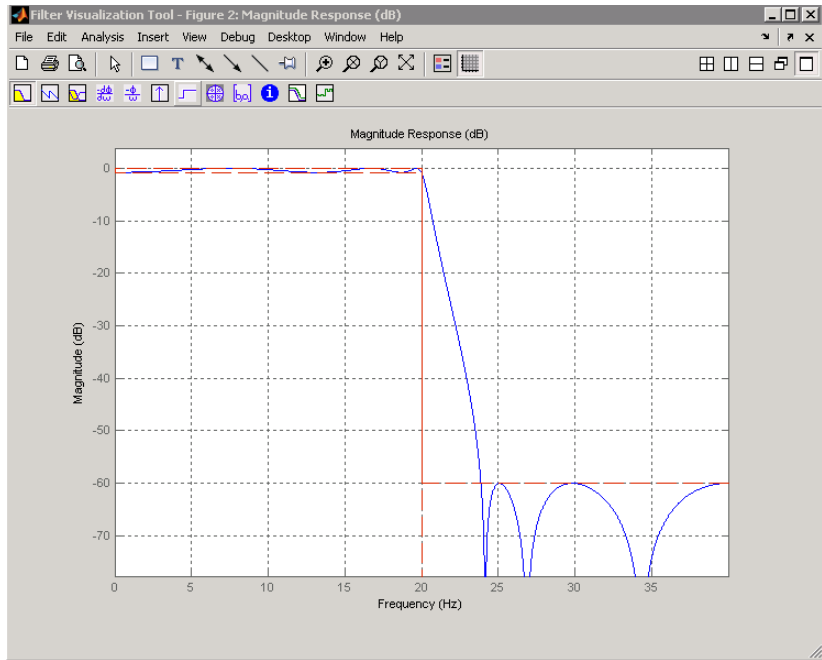
Description   `hd = design(d,'equiripple')` designs an equiripple FIR digital
filter or multirate filter using the specifications supplied in the object
`d`. Equiripple filter designs minimize the maximum ripple in the
passbands and stopbands.

hd is either a `dfilt` object (a single-rate digital filter) or an `mfilt` object
(a multirate digital filter) depending on the `Specification` property
of the filter specification object `d` and the specifications object type —
halfband or interpolator.

When you use `equiripple` with Nyquist filter specification objects, you
might encounter design cases where the filter design does not converge.
Convergence errors occur mostly at large filter orders, or small
transition widths, or large stopband attenuations. These specifications,
alone or combined, can cause design failures. For more information,
refer to `fdesign.nyquist` in the online Help system.

`hd = design(d,'equiripple',designoption,value,designoption,
...value,...)` returns an equiripple FIR filter where you specify
design options as input arguments.

To determine the available design options, use `designopts` with the
specification object and the design method as input arguments as
shown.

```
designopts(d,'method')
```

For complete help about using `equiripple`, refer to the command line
help system. For example, to get specific information about using
`equiripple` with `d`, the specification object, enter the following at the
MATLAB prompt.

```
help(d,'equiripple')
```

equiripple

Examples

Here is an example of designing a single-rate equiripple filter from a halfband filter specification object. Notice the help command used to learn about the options for the specification object and method.

```
d = fdesign.halfband('tw,ast',0.1,80);
designmethods(d)
help(d,'equiripple')
designopts(d,'equiripple')
hd = design(d,'equiripple','stopbandshape','flat');
fvtool(hd);
```

Displaying the filter in FVTool shows the equiripple nature of the filter.
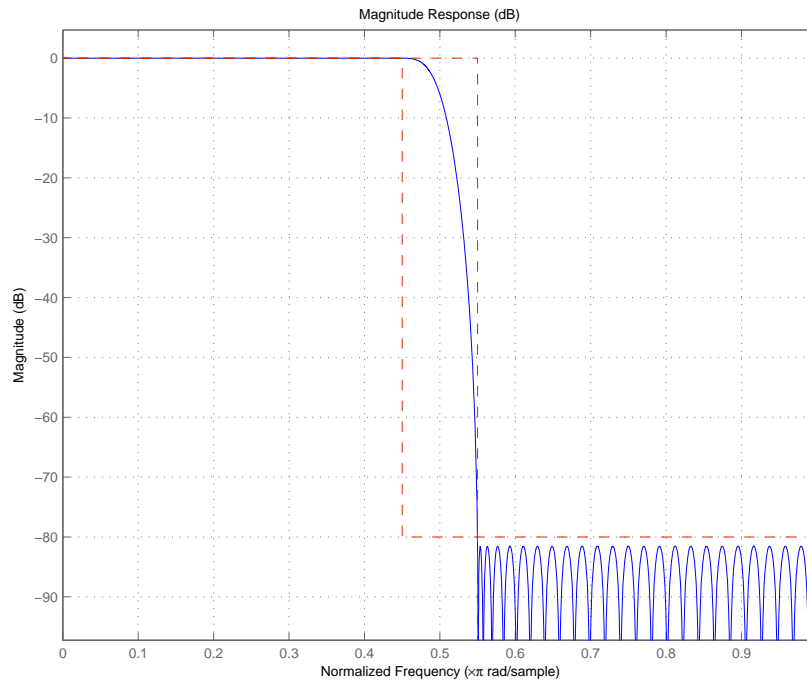
equiripple also designs multirate filters. This example generates a halfband interpolator filter.

```
d = fdesign.interpolator(2); % Interpolation factor = 2.
hd = design(d,'equiripple');
```

This final example designs an equiripple filter with a direct-form structure by specifying the **filterstructure** argument. To set the design options for the filter, use the designopts method and options object opts.

```
d = fdesign.lowpass('fp,fst,ap,ast');
opts=designopts(d,'equiripple')
opts.FilterStructure='dffirt'
opts.MinPhase=1;
opts.DensityFactor=20;
opts
hd=design(d,'equiripple',opts)
```

Note The MaxPhase design option for equripple FIR filters is currently only available for lowpass, highpass, bandpass, and bandstop filters.

See Also   fdesign.nyquist | firls | kaiserwin

fcfwrite

Purpose   Write file containing filter coefficients

Syntax
```
fcfwrite(h)
fcfwrite(h,filename)
fcfwrite(...,'fmt')
```

Description   fcfwrite(h) writes a filter coefficient ASCII file in a folder you choose, or your current MATLAB working folder. h can be a single filter object or a vector of filter objects. On execution, fcfwrite opens the **Export Filter Coefficients to .FCF File** dialog box to let you assign a file name for the output file. You can choose the destination folder within this dialog as well.

The default file name is untitled.fcf. When you have DSP System Toolbox software, you can use fcfwrite(h) to write filter coefficient files for multirate filters, adaptive filters, and discrete-time filters.

fcfwrite(h,filename) writes the filter coefficients and general information to a text file called filename in your present MATLAB working folder and opens the file in the MATLAB editor for you to review or modify.

If you do not include a file extension in filename, fcfwrite adds the extension fcf to filename.

fcfwrite(...,'fmt') writes the filter coefficients in the format specified by the input argument fmt. Valid fmt values are hex for hexadecimal, dec for decimal, or bin for binary representation of the filter coefficients.

Examples   To demonstrate fcfwrite, create a fixed-point IIR filter at the command line, and then write the filter coefficients to a file named iirfilter.fcf.

```
d=fdesign.lowpass;
hd=design(d,'butter');
set(hd,'arithmetic','fixed');
fcfwrite(hd,'iirfilter.fcf');
```

Here is the output from `fcfwrite` as it appears in the MATLAB editor. Not shown here is the filename — `iirfilter.fcf` as specified and some comments at the top of the file.

```
%
%
% Coefficient Format: Decimal
%
% Discrete-Time IIR Filter (real)
% -------------------------------
% Filter Structure   : Direct-Form II, Second-Order
%                          Sections
% Number of Sections : 13
% Stable             : Yes
% Linear Phase       : No
% Arithmetic         : fixed
% Numerator          : s16,13 -> [-4 4)
% Denominator        : s16,14 -> [-2 2)
% Scale Values       : s16,14 -> [-2 2)
% Input              : s16,15 -> [-1 1)
% Section Input      : s16,8 -> [-128 128)
% Section Output     : s16,10 -> [-32 32)
% Output             : s16,10 -> [-32 32)
% State              : s16,15 -> [-1 1)
% Numerator Prod     : s32,28 -> [-8 8)
% Denominator Prod   : s32,29 -> [-4 4)
% Numerator Accum    : s40,28 -> [-2048 2048)
% Denominator Accum  : s40,29 -> [-1024 1024)
% Round Mode         : convergent
% Overflow Mode      : wrap
% Cast Before Sum    : true


SOS matrix:
1  2  1  1  -0.22222900390625  0.88262939453125
1  2  1  1  -0.19903564453125  0.68621826171875
1  2  1  1  -0.18060302734375  0.5303955078125
```

```
1  2  1  1  -0.1658935546875    0.40570068359375
1  2  1  1  -0.154052734375     0.305419921875
1  2  1  1  -0.14453125         0.22479248046875
1  2  1  1  -0.136962890625     0.16015625
1  2  1  1  -0.13092041015625   0.10906982421875
1  2  1  1  -0.126220703125     0.06939697265625
1  2  1  1  -0.12274169921875   0.0399169921875
1  2  1  1  -0.12030029296875   0.01947021484375
1  2  1  1  -0.118896484375     0.0074462890625
1  1  0  1  -0.0592041015625    0

Scale Values:
0.41510009765625
0.371826171875
0.33746337890625
0.3099365234375
0.287841796875
0.27008056640625
0.25579833984375
0.2445068359375
0.23577880859375
0.22930908203125
0.22479248046875
0.22216796875
0.47039794921875
1
```

To write two or more filters out to one file, provide the filters as a vector to `fcfwrite`:

```
fcfwrite([hd hd1 hd2])
```

See Also   adaptfilt | mfilt | dfilt

Purpose   Open Filter Design and Analysis Tool

Syntax   `fdatool`

Description `fdatool` opens the Filter Design and Analysis Tool (FDATool). Use this tool to:

- Design filters

- Quantize filters (with DSP System Toolbox software installed)

- Analyze filters

- Modify existing filter designs

- Create multirate filters (with DSP System Toolbox software installed)

- Realize Simulink models of quantized, direct-form, FIR filters (with DSP System Toolbox software installed)

- Perform digital frequency transformations of filters (with DSP System Toolbox software installed)

Refer to "Use FDATool with DSP System Toolbox Software" for more information about using the analysis, design, and quantization features of FDATool. For general information about using FDATool, refer to "FDATool" in Signal Processing Toolbox documentation.

When you open FDATool and you have DSP System Toolbox software installed, FDATool incorporates features that are added by DSP System Toolbox software. With DSP System Toolbox software installed, FDATool lets you design and analyze quantized filters, as well as convert quantized filters to various filter structures, transform filters, design multirate filters, and realize models of filters.
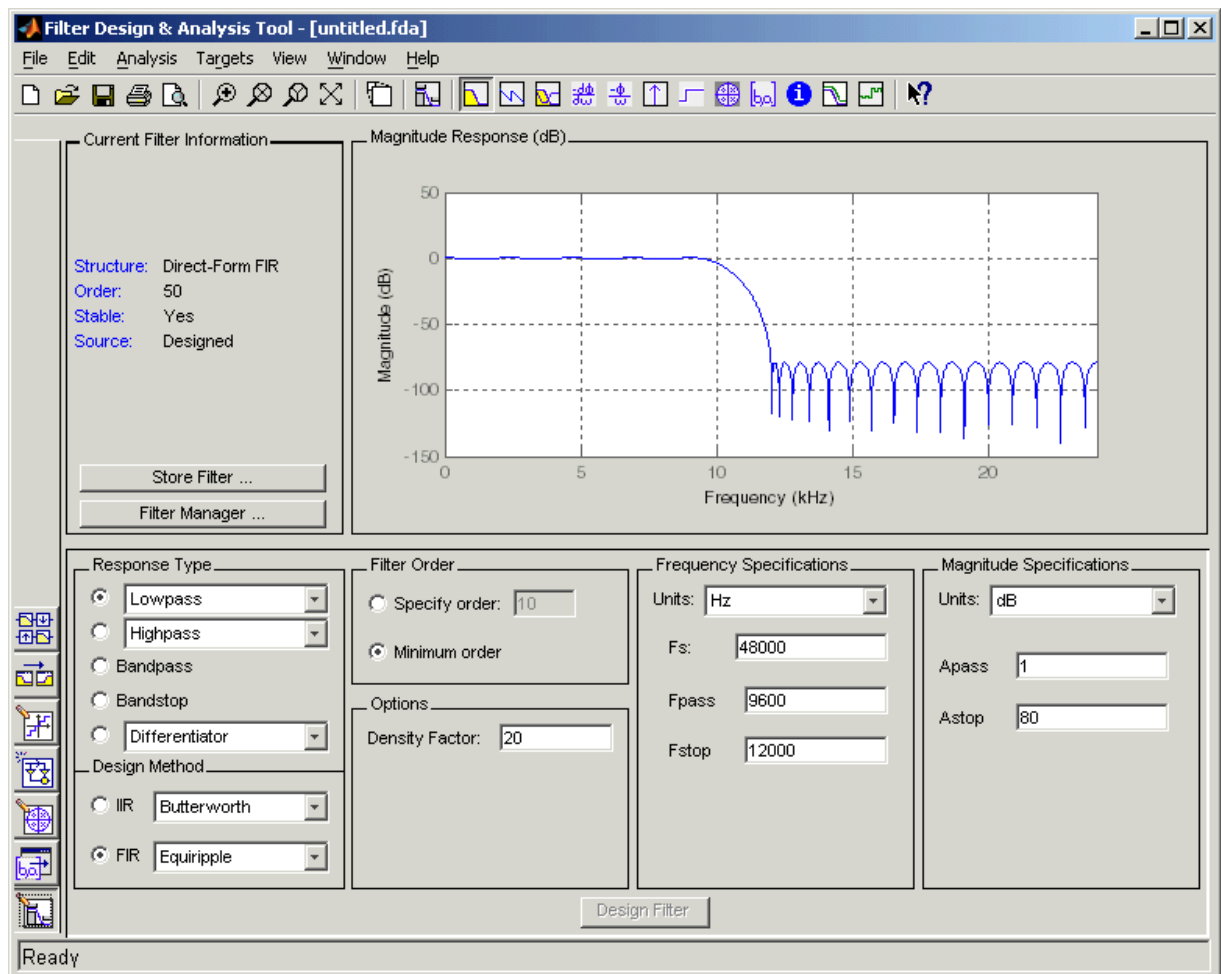
fdatool

Use the buttons on the sidebar to configure the design area to use various tools in FDATool.

Set Quantization Parameters — provides access to the properties of the quantizers that compose a quantized filter. When you click **Set Quantization Parameters**, you see FDATool displaying the

quantization options at the bottom of the dialog box (the design area), as shown in the figure.

Transform Filter — clicking this button opens the *Frequency Transformations* pane so you can use digital frequency transformations to change the magnitude response of your filter.

Create a multirate filter — clicking this button switches FDATool to multirate filter design mode so you can design interpolators, decimators, and fractional rate change filters.

Realize Model — starting from your quantized, direct-form, FIR filter, clicking this button creates a Simulink model of your filter structure in new model window.
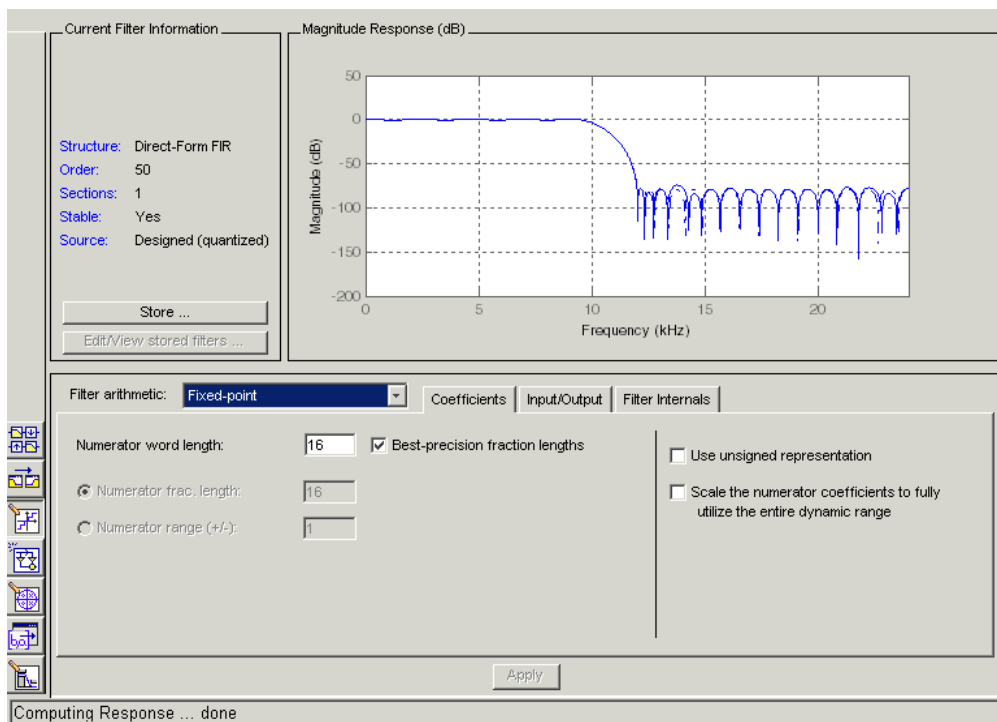
Other options in the menu bar let you convert the filter structure to a new structure, change the order of second-order sections in a filter, or change the scaling applied to the filter, among many possibilities.

Tips
By incorporating many advanced filter design methods from DSP System Toolbox software, FDATool provides more design methods than the SPTool Filter Designer.

See Also
fdatool | fvtool | sptool

Purpose   Filter specification object

Syntax
```
d = fdesign.response
d = fdesign.response(spec)
d = fdesign.response(...,Fs)
d = fdesign.response(...,magunits)
```

Description   **Filter Specification Objects**

d = fdesign.*response* returns a filter specification object d, of filter response *response*. To create filters from d, use one of the design methods listed in "Using Filter Design Methods with Specification Objects" on page 7-484

Note Several of the filter response types described below are only available if your installation includes the DSP System Toolbox. The DSP System Toolbox significantly expands the functionality available for the specification, design, and analysis of filters.

Here is how you design filters using fdesign.

1 Use fdesign.*response* to construct a filter specification object.

2 Use designmethods to determine which filter design methods work for your new filter specification object.

3 Use design to apply your filter design method from step 2 to your filter specification object to construct a filter object.

4 Use FVTool to inspect and analyze your filter object.

fdesign

> **Note** fdesign does not create filters. fdesign returns a filter specification object that contains the specifications for a filter, such as the passband cutoff or attenuation in the stopband. To design a filter hd from a filter specification object d, use d with a filter design method such as butter —hd = design(d,'butter').

For more guidance about using fdesign, refer to the examples in "Getting Started". Alternatively, type the following at the MATLAB prompt for more information:

```
help fdesign
```

response can be one of the entries in the following table that specify the filter response desired, such as a bandstop filter or an interpolator.

fdesign Response String	Description
arbgrpdelay	fdesign.arbgrpdelay creates an object to specify allpass arbitrary group delay filters. Requires the DSP System Toolbox
arbmag	fdesign.arbmag creates an object to specify IIR filters that have arbitrary magnitude responses defined by the input arguments.
arbmagnphase	fdesign.arbmagnphase creates an object to specify IIR filters that have arbitrary magnitude and phase responses defined by the input arguments. Requires the DSP System Toolbox.
audioweighting	fdesign.audioweighting creates a filter specification object for audio weighting filters. The supported audio weighting types are: A, C, C-message, ITU-T 0.41, and ITU-R 468-4 weighting. Requires the DSP System Toolbox

fdesign Response String	Description
bandpass	fdesign.bandpass creates an object to specify bandpass filters.
bandstop	fdesign.bandstop creates an object to specify bandstop filters.
ciccomp	fdesign.ciccomp creates an object to specify filters that compensate for the CIC decimator or interpolator response curves. Requires the DSP System Toolbox.
comb	fdesign.comb creates an object to specify a notching or peaking comb filter. Requires the DSP System Toolbox.
decimator	fdesign.decimator creates an object to specify decimators. Requires the DSP System Toolbox
differentiator	fdesign.differentiator creates an object to specify an FIR differentiator filter.
fracdelay	fdesign.fracdelay creates an object to specify fractional delay filters. Requires the DSP System Toolbox.
halfband	fdesign.halfband creates an object to specify halfband filters. Requires the DSP System Toolbox.
highpass	fdesign.highpass creates an object to specify highpass filters.
hilbert	fdesign.hilbert creates an object to specify an FIR Hilbert transformer.
interpolator	fdesign.interpolator creates an object to specify interpolators. Requires the DSP System Toolbox.

fdesign Response String	Description
isinchp	fdesign.isinchp creates an object to specify an inverse sinc highpass filter. Requires the DSP System Toolbox.
isinclp	fdesign.isinclp creates an object to specify an inverse sinc lowpass filters. Requires the DSP System Toolbox.
lowpass	fdesign.lowpass creates an object to specify lowpass filters.
notch	fdesign.notch creates an object to specify notch filters. Requires the DSP System Toolbox.
nyquist	fdesign.nyquist creates an object to specify nyquist filters. Requires the DSP System Toolbox.
octave	fdesign.octave creates an object to specify octave and fractional octave filters. Requires the DSP System Toolbox.
parameq	fdesign.parameq creates an object to specify parametric equalizer filters. Requires the DSP System Toolbox.
peak	fdesign.peak creates an object to specify peak filters. Requires the DSP System Toolbox.
polysrc	fdesign.polysrc creates an object to specify polynomial sample-rate converter filters. Requires the DSP System Toolbox.
pulseshaping	fdesign.pulseshaping creates an object to specify pulse-shaping filters.
rsrc	fdesign.rsrc creates an object to specify rational-factor sample-rate convertors. Requires the DSP System Toolbox.

Use the `doc fdesign.`*`response`* syntax at the MATLAB prompt to get help on a specific structure. Using `doc` in a syntax like

```
doc fdesign.lowpass
doc fdesign.bandstop
```

gets more information about the lowpass or bandstop structure objects.

Each `response` has a property `Specification` that defines the specifications to use to design your filter. You can use defaults or specify the `Specification` property when you construct the specifications object.

With the strings for the `Specification` property, you provide filter constraints such as the filter order or the passband attenuation to use when you construct your filter from the specification object.

Properties   `fdesign` returns a filter specification object. Every filter specification object has the following properties.

Property Name	Default Value	Description
Response	Depends on the chosen type	Defines the type of filter to design, such as an interpolator or bandpass filter. This is a read-only value.
Specification	Depends on the chosen type	Defines the filter characteristics used to define the desired filter performance, such as the cutoff frequency Fc or the filter order N.

fdesign

Property Name	Default Value	Description
Description	Depends on the filter type you choose	Contains descriptions of the filter specifications used to define the object, and the filter specifications you use when you create a filter from the object. This is a read-only value.
NormalizedFrequency	Logical true	Determines whether the filter calculation uses normalized frequency from 0 to 1, or the frequency band from 0 to Fs/2, the sampling frequency. Accepts either true or false without single quotation marks. Audio weighting filters do not support normalized frequency.

In addition to these properties, filter specification objects may have other properties as well, depending on whether they design dfilt objects or mfilt objects.

Added Properties for mfilt Objects	Description
DecimationFactor	Specifies the amount to decrease the sampling rate. Always a positive integer.
InterpolationFactor	Specifies the amount to increase the sampling rate. Always a positive integer.
PolyphaseLength	Polyphase length is the length of each polyphase subfilter that composes the decimator or interpolator or rate-change

Added Properties for mfilt Objects	Description
	factor filters. Total filter length is the product of `pl` and the rate change factors. `pl` must be an even integer.

`d = fdesign.`*`response`*`(spec)`. In `spec`, you specify the variables to use that define your filter design, such as the passband frequency or the stopband attenuation. The specifications are applied to the filter design method you choose to design your filter.

For example, when you create a default lowpass filter specification object, `fdesign.lowpass` sets the passband frequency `Fp`, the stopband frequency `Fst`, the stopband attenuation `Ast`, and the passband ripple `Ap` :

```
H = fdesign.lowpass
% Without a terminating semicolon
% the filter specifications are displayed
```

The default specification `'Fp,Fst,Ap,Ast'` is only one of the possible specifications for `fdesign.lowpass`. To see all available specifications:

```
H = fdesign.lowpass;
set(H,'specification')
```

The DSP System Toolbox software supports all available specification strings. The Signal Processing Toolbox supports a subset of the specification strings. See the reference pages for the filter specification object to determine which specification strings your installation supports.

One important note is that the specification string you choose determines which design methods apply to the filter specifications object.

Specifications that do not contain the filter order result in minimum order designs when you invoke the `design` method:

```
d = fdesign.lowpass;
% Specification is Fp,Fst,Ap,Ast
Hd = design(d,'equiripple');
length(Hd.Numerator) % returns 43
% Filter order is 42
fvtool(Hd) %view magnitude
```

d = fdesign.*response*(...,Fs) specifies the sampling frequency in Hz to use in the filter specifications. The sampling frequency is a scalar trailing all other input arguments. If you specify a sampling frequency, all frequency specifications are in Hz.

d = fdesign.*response*(...,magunits) specifies the units for any magnitude specification you provide in the input arguments. magunits can be one of the following strings:

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in decibels
- 'squared' — specify the magnitude in power units

When you omit the magunits argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Using Filter Design Methods with Specification Objects

After you create a filter specification object, you use a filter design method to implement your filter with a selected algorithm. Use designmethods to determine valid design methods for your filter specification object.

```
d = fdesign.lowpass('N,Fc,Ap,Ast',10,0.2,0.5,40);
designmethods(d)
% Design FIR equiripple filter
hd = design(d,'equiripple');
```

When you use any of the design methods without providing an output argument, the resulting filter design appears in FVTool by default.

Along with filter design methods, `fdesign` works with supporting methods that help you create filter specification objects or determine which design methods work for a given specifications object.

Supporting Function	Description
`setspecs`	Set all of the specifications simultaneously.
`designmethods`	Return the design methods.
`designopts`	Return the input arguments and default values that apply to a specifications object and method

You can set filter specification values by passing them after the `Specification` argument, or by passing the values without the `Specification` string.

Filter object constructors take the input arguments in the same order as `setspecs` and the order in the strings for `Specification`. Enter `doc setspecs` at the prompt for more information about using `setspecs`.

When the first input to `fdesign` is not a valid `Specification` string like 'n,fc', `fdesign` assumes that the input argument is a filter specification and applies it using the default `Specification` string —fp,fst,ap,ast for a lowpass object, for example.

Examples   The following examples require only the Signal Processing Toolbox.

Example 1–Bandstop Filter

A bandstop filter specification object for data sampled at 8 kHz. The stopband between 2 and 2.4 kHz is attenuated at least 80 dB:

```
H = fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2',...
          1600,2000,2400,2800,1,80,1,8000);
```

Example 2–Lowpass Filter

A lowpass filter specification object for data sampled at 10 kHz. The passband frequency is 500 Hz and the stopband frequency is 750 Hz.

The passband ripple is set to 1 dB and the required attenuation in the stopband is 80 dB.

```
H = fdesign.lowpass('Fp,Fst,Ap,Ast',500,750,1,80,10000);
```

Example 3—Highpass Filter

A default highpass filter specification object.

```
H = fdesign.highpass % Creates specifications object.
H.Description
```

Notice the correspondence between the property values in `Specification` and `Description` — in `Description` you see in words the definitions of the variables shown in `Specification`.

Example 4—Filter Specification and Design

Lowpass Butterworth filter specification object

Use a filter specification object to construct a lowpass Butterworth filter with default `Specification` `'Fp,Fst,Ap,Ast'`. Set the passband edge frequency to 0.4π radians/sample, a stopband frequency of 0.5π radians/sample, a passband ripple of 1 dB, and 80 dB of stopband attenuation.

```
d = fdesign.lowpass(0.4,0.5,1,80);
```

Determine which design methods apply to `d`.

```
designmethods(d)
```

You can use `d` and the `butter` design method to design a Butterworth filter.

```
hd = design(d,'butter','matchexactly','passband');
fvtool(hd);
```

The resulting filter magnitude response shown by FVTool appears in the following figure.
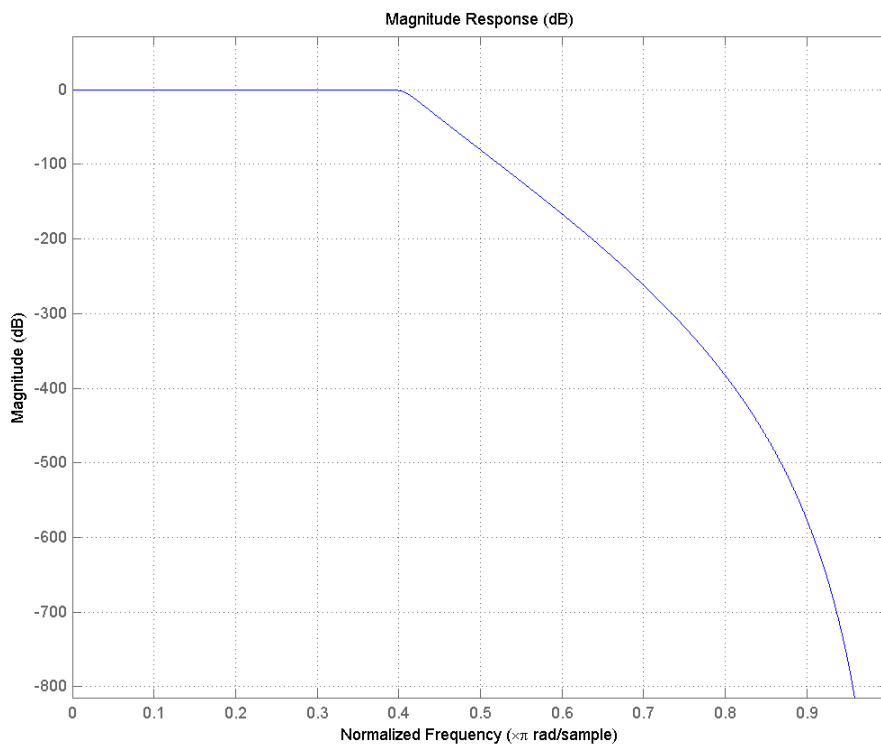
If you have the DSP System Toolbox software installed, the preceding figure appears with the filter specification mask.

See Also   designmethods | designopts | fdatool | filterbuilder | fvtool

fdesign.arbgrpdelay

Purpose   Arbitrary group delay filter specification object

Syntax
```
D = fdesign.arbgrpdelay(SPEC)
D = fdesign.arbgrpdelay(SPEC,SPEC1,SPEC2,...)
D = fdesign.arbgrpdelay(N,F,Gd)
D = fdesign.arbgrpdelay(...,Fs)
```

Description   Arbitrary group delay filters are allpass filters you can use for correcting phase distortion introduced by other filters. `fdesign.arbgrpdelay` uses an iterative least *p*-th norm optimization procedure to minimize the phase response error [1].

D = fdesign.arbgrpdelay(SPEC) specifies an allpass arbitrary group delay filter with the `Specification` property set to `SPEC`. See "Input Arguments" on page 7-489 for a description of supported specifications.

D = fdesign.arbgrpdelay(SPEC,SPEC1,SPEC2,...) initializes the allpass arbitrary group delay filter specification object with specifications `SPEC1,SPEC2,...`. See SPEC for a description of supported specifications.

D = fdesign.arbgrpdelay(N,F,Gd) specifies an allpass arbitrary group delay filter. The filter order is equal to `N`, frequency vector equal to `F`, and group delay vector equal to `Gd`. See SPEC for a description of the filter order, frequency vector, and group delay vector inputs. See the example Allpass Filter with Arbitrary Group Delay for an example using this syntax.

D = fdesign.arbgrpdelay(...,Fs) specifies the sampling frequency in hertz as a trailing scalar. If you do not specify a sampling frequency, all frequencies are normalized frequencies and group delay values are in samples. If you specify a sampling frequency, group delay values are in seconds.

Tips   If your arbitrary group delay design produces the error `Poorly conditioned Hessian matrix`, attempt one or more of the following:

- Set the `MaxPoleRadius` IIR lp norm design option to some number less than 1. Set this option when you `design` your filter with the syntax:

  ```
  design(d,'iirlpnorm','MaxPoleRadius',0.95)
  ```

  See the Frequency Dispersion and Multiband Delay Equalization examples for the use of the `MaxPoleRadius` design option.

- Reduce the order of your filter design.

Input Arguments

SPEC

Specification string. `SPEC` is one of the following two strings. The entries are not case sensitive.

- `'N,F,Gd'`

- `'N,B,F,Gd'`

The string entries are defined as follows:

- `N` — Filter order. This value must be an even positive integer. The numerator and denominator orders are both equal to `N`. "Allpass Systems" on page 7-491 explains why the numerator and denominator filter orders are equal and the order must be even in `fdesign.arbgrpdelay`.

- `F` — Frequency vector for the group delay specifications. The elements of the frequency vector must increase monotonically. If you do not specify a sampling frequency, `Fs`, in hertz, the frequencies are normalized frequencies. For a single-band design, the first element of the normalized frequency vector must be zero and the last element must be 1. These correspond to 0 and π radians/sample respectively. For multiband designs, the union of the frequency vectors must range from [0,1].

  If you specify a sampling frequency, `Fs`, the first element of the frequency vector in a single-band design must be 0. The last element must be the Nyquist frequency, Fs/2. For multiband designs, the union of the frequency vectors must range from [0,Fs/2].

fdesign.arbgrpdelay

- Gd — Group delay vector. A vector with nonnegative elements equal in length to the frequency vector, F. The elements of Gd specify the nonnegative group delay at the corresponding element of the frequency vector, F.

  If you do not specify a sampling frequency, Fs, in Hertz, the group delays are in samples. If you specify a sampling frequency, the group delays are in seconds.

- B — Number of frequency bands. If you use this specification, you must specify a frequency and group delay vector for each band. The union of the frequency vectors must range from [0,1] in normalized frequency, or [0,Fs/2] when a sampling frequency is specified. The elements in the union of the frequency bands must be monotonically increasing.

  For example:

  ```
  filtorder = 14;
  freqband1 = [0 0.1 0.4]; grpdelay1 = [1 2 3];
  freqband2 = [0.5 0.8 1]; grpdelay2 = [3 2 1];
  D = fdesign.arbgrpdelay('N,B,F,Gd',filtorder,2,freqband1,grpdelay1,freq
  ```

  Default: `'N,F,Gd'`

Fs

Sampling frequency. Specify the sampling frequency as a trailing positive scalar after all other input arguments. Specifying a sampling frequency forces the group delay units to be in seconds. If you specify a sampling frequency, the first element of the frequency vector must be 0. The last element must be the Nyquist frequency, Fs/2.

Output Arguments

D

Filter specification object. An allpass arbitrary group delay filter specification object containing the following modifiable properties: Specification, NormalizedFrequency, FilterOrder, Frequencies, and GroupDelay.

Use the `normalizefreq` method to change the `NormalizedFrequency` property after construction.

Definitions   ### Group Delay in Discrete-Time Filter Design

The frequency response of a rational discrete-time filter is:

$$H(e^{j\omega}) = \frac{B(e^{j\omega})}{A(e^{j\omega})}$$

The argument of the frequency response as a function of the angle, ω, is referred to as the *phase response*.

The negative of the first derivative of the argument with respect to ω is the group delay.

$$\tau(\omega) = -\frac{d}{d\omega} Arg(H(e^{j\omega}))$$

Systems with nonlinear phase responses have nonconstant group delay, which causes dispersion of the frequency components of the signal. You may not want this phase distortion even if the magnitude distortion introduced by the filter produces the desired effect. See Frequency Dispersion for an illustration of frequency dispersion resulting from nonconstant group delay.

In these cases, you can cascade the frequency-selective filter with an allpass filter that compensates for the group delay. This process is often referred to as *delay equalization*.

Allpass Systems

The general form of an allpass system function with a real-valued impulse response is:

$$H_{ap}(z) = \prod_{k=1}^{M} \frac{z^{-1} - d_k}{1 - d_k z^{-1}} \prod_{k=1}^{N} \frac{(z^{-1} - c_k)(z^{-1} - c_k^*)}{(1 - c_k z^{-1})(1 - c_k^* z^{-1})}$$

where the d_k denote the real-valued poles and the c_k denote the complex-valued poles, which occur in conjugate pairs.

The preceding equation demonstrates that allpass systems with real-valued impulse responses have *2N+M* zeros and poles. The poles and zeros occur in pairs with reciprocal magnitudes. The filter order is always the same for the numerator and denominator.

Because `fdesign.arbgrpdelay` uses robust second-order section (biquad) filter structures to implement the allpass arbitrary group delay filter, the filter order must be even.

Examples   Construct a signal consisting of two discrete-time windowed sinusoids (wave packets) with disjoint time support to illustrate frequency dispersion. One discrete-time sinusoid has a frequency of π/2 radians/sample and the other has a frequency of π/4 radians/sample. There are 9 periods of the higher–frequency sinusoid that precede 5 periods of the lower–frequency signal.

Create the signal.

```
x = zeros(300,1);
x(1:36) = cos(pi/2*(0:35)).*hamming(36)';
x(40:40+39) = cos(pi/4*(0:39)).*hamming(40)';
```

Create an arbitrary group delay filter that delays the higher–frequency wave packet by approximately 100 samples.

```
N = 18;
f = 0:.1:1;
gd = ones(size(f));
% Delay pi/2 radians/sample by 100 samples
gd(6) = 100;
d = fdesign.arbgrpdelay(N,f,gd);
Hd = design(d,'iirlpnorm','MaxPoleRadius',0.9);
% Visualize the group delay
fvtool(Hd,'analysis','grpdelay');
```
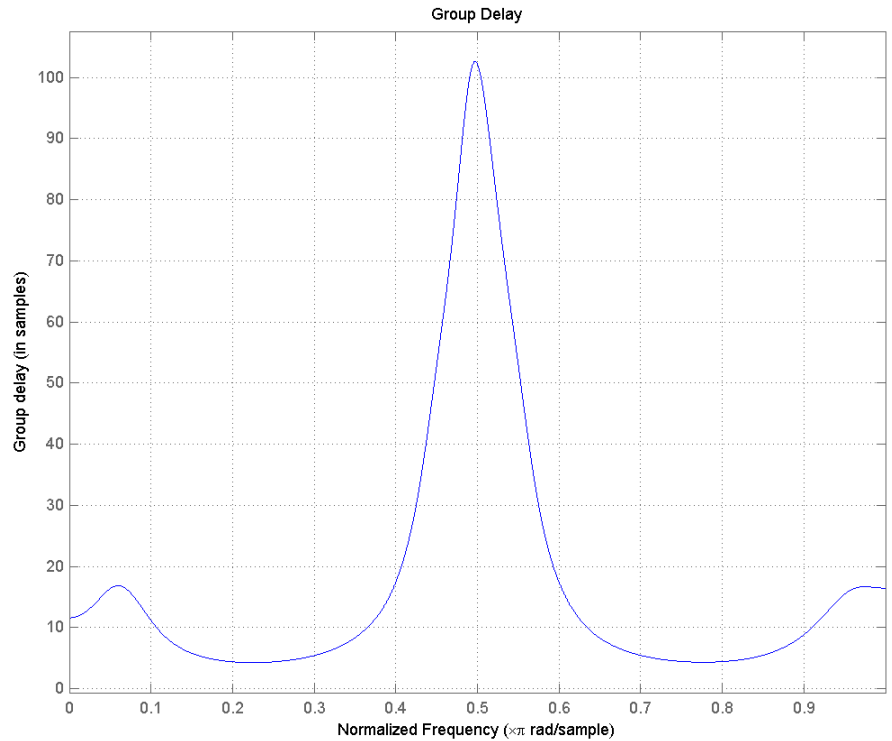
Filter the input signal with the arbitrary group delay filter and illustrate the frequency dispersion. The high–frequency wave packet, which initially preceded the low–frequency wave packet, now occurs later because of the nonconstant group delay.

```
y = filter(Hd,x);
subplot(211)
plot(x); title('Input Signal');
grid on; ylabel('Amplitude');
subplot(212);
plot(y); title('Output Signal'); grid on;
```

```
xlabel('Samples'); ylabel('Amplitude');
```
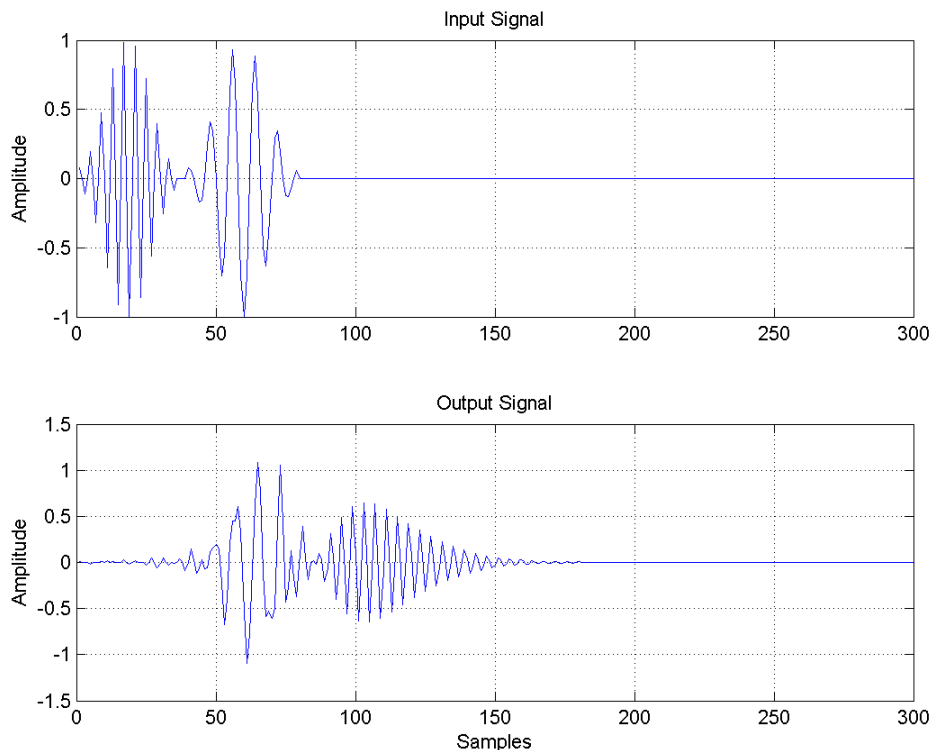

Design an allpass filter with an arbitrary group delay.

```
N = 10;
f = [0 0.02 0.04 0.06 0.08 0.1 0.25 0.5 0.75 1];
g = [5 5 5 5 5 5 4 3 2 1];
w = [2 2 2 2 2 2 1 1 1 1];
hgd = fdesign.arbgrpdelay(N,f,g);
Hgd = design(hgd,'iirlpnorm','Weights',w,'MaxPoleRadius',0.95);
```

```
fvtool(Hgd,'Analysis','grpdelay') ;
```
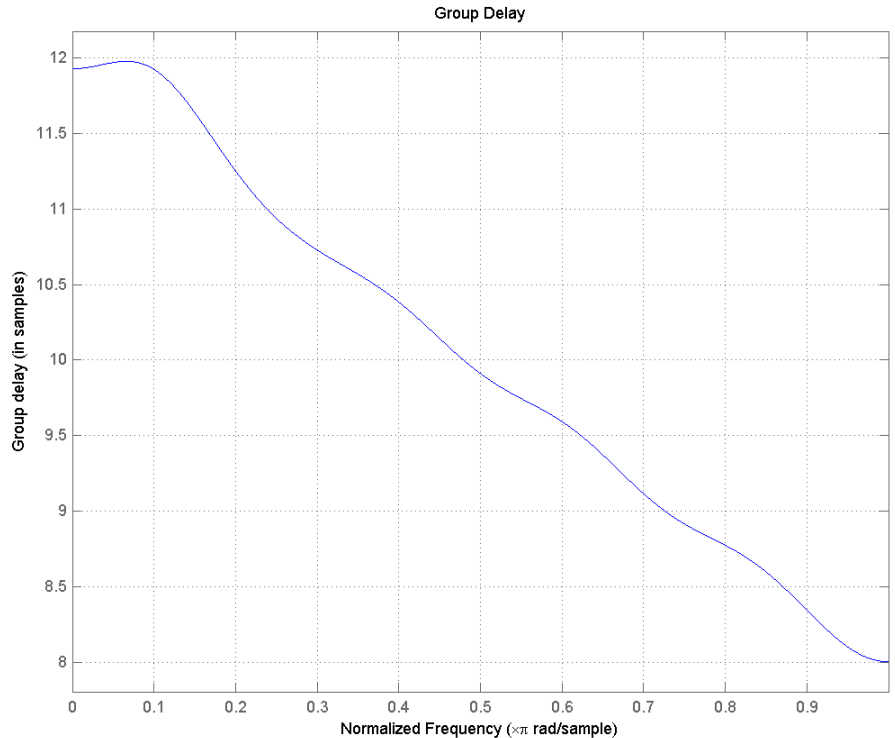

Perform multiband delay equalization outside the stopband.

```
Fs = 1e3;
Hcheby2 = design(fdesign.bandstop('N,Fst1,Fst2,Ast',10,150,400,60,Fs);
f1 = 0.0:0.5:150; % Hz
g1 = grpdelay(Hcheby2,f1,Fs).'/Fs; % seconds
f2 = 400:0.5:500; % Hz
g2 = grpdelay(Hcheby2,f2,Fs).'/Fs; % seconds
```

```
maxg = max([g1 g2]);
% Design an arbitrary group delay allpass filter to equalize the group
% delay of the bandstop filter. Use an order 18 multiband design and spec
% two bands.
hgd = fdesign.arbgrpdelay('N,B,F,Gd',18,2,f1,maxg-g1,f2,maxg-g2,Fs);
Hgd = design(hgd,'iirlpnorm','MaxPoleRadius',0.95);
Hcascade = cascade(Hcheby2,Hgd);
hft = fvtool(Hcheby2,Hgd,Hcascade,'Analysis','grpdelay','Fs',Fs);
    legend(hft,'Original Bandstop Filter','Allpass Arbitrary Group Delay
    'Delay Equalization', 'Location','North');
```
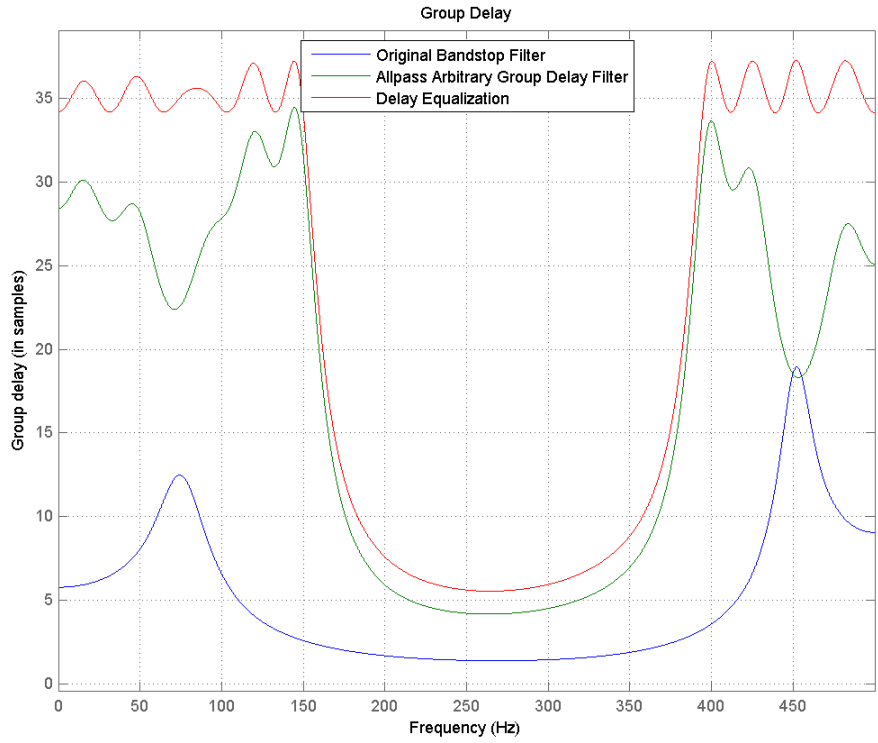
Algorithms   `fdesign.arbgrpdelay` uses a least *p*-th norm iterative optimization described in [1].

References   [1] Antoniou, A. *Digital Signal Processing: Signals, Systems, and Filters.*, New York:McGraw-Hill, 2006, pp. 719–771.

Alternatives   `iirgrpdelay` — Returns an allpass arbitrary group delay filter. The `iirgrpdelay` function returns the numerator and denominator coefficients. This behavior differs from that of `fdesign.arbgrpdelay`, which returns the filter in second-order sections. `iirgrpdelay` accepts only normalized frequencies.

See Also   `fdesign` | `design` | `iirgrpdelay`

How To   • "Design a Filter in Fdesign — Process Overview"

fdesign.arbmag

Purpose	Arbitrary response magnitude filter specification object

Syntax

```
D= fdesign.arbmag
D= fdesign.arbmag(SPEC)
D = fdesign.arbmag(SPEC,specvalue1,specvalue2,...)
D = fdesign.arbmag(specvalue1,specvalue2,specvalue3)
D = fdesign.arbmag(...,Fs)
```

Description D= fdesign.arbmag constructs an arbitrary magnitude filter specification object D.

D= fdesign.arbmag(SPEC) initializes the Specification property to SPEC. The input argument SPEC must be one of the strings shown in the following table. Specification strings are not case sensitive.

Note Specifications strings marked with an asterisk require the DSP System Toolbox software.

- 'N,F,A' — Single band design (default)
- 'F,A,R' — Single band minimum order design *
- 'N,B,F,A' — Multiband design
- 'N,B,F,A,C' — Constrained multiband design *
- 'B,F,A,R' — Multiband minimum order design *
- 'Nb,Na,F,A' — Single band design *
- 'Nb,Na,B,F,A' — Multiband design *

The string entries are defined as follows:

- A — Amplitude vector. Values in A define the filter amplitude at frequency points you specify in f, the frequency vector. If you use A, you must use F as well. Amplitude values must be real. For complex values designs, use fdesign.arbmagnphase.

- B — Number of bands in the multiband filter

- C — Constrained band flag. This enables you to constrain the passband ripple in your multiband design. You cannot constrain the passband ripple in all bands simultaneously.

- F — Frequency vector. Frequency values in specified in F indicate locations where you provide specific filter response amplitudes. When you provide F, you must also provide A.

- N — Filter order for FIR filters and the numerator and denominator orders for IIR filters.

- Nb — Numerator order for IIR filters

- Na — Denominator order for IIR filter designs

- R — Ripple

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

Specifying Frequency and Amplitude Vectors

F and A are the input arguments you use to define the filter response desired. Each frequency value you specify in F must have a corresponding response value in A. The following table shows how F and A are related.

Define the frequency vector F as [0 0.25 0.3 0.4 0.5 0.6 0.7 0.75 1.0]

Define the response vector A as [1 1 0 0 0 0 0 1 1]

These specifications connect F and A as shown here:

F (Normalized Frequency)	A (Response Desired at F)
0	1
0.25	1
0.3	0
0.4	0

F (Normalized Frequency)	A (Response Desired at F)
0.5	0
0.6	0
0.7	0
0.75	1
1.0	1

Different specifications can have different design methods available. Use `designmethods` to get a list of design methods available for a given specification string and filter specification object.

Use `designopts` to get a list of design options available for a filter specification object and a given design method. Enter `help(D,METHOD)` to get detailed help on the available design options for a given design method.

`D = fdesign.arbmag(SPEC,specvalue1,specvalue2,...)` initializes the specifications with `specvalue1`, `specvalue2`. Use `get(D,'Description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ... `specvalueN`.

`D = fdesign.arbmag(specvalue1,specvalue2,specvalue3)` uses the default specification string `'N,F,A'`, setting the filter order, filter frequency vector, and the amplitude vector to the values `specvalue1`, `specvalue2`, and `specvalue3`.

`D = fdesign.arbmag(...,Fs)` specifies the sampling frequency in Hz. All other frequency specifications are also assumed to be in Hz when you specify `Fs`.

Examples   **Design of a multiband arbitrary-magnitude filter**

Use fdesign.arbmag to design a 3–band filter.

Use the given frequency and amplitude vectors in "Specifying Frequency and Amplitude Vectors" on page 7-499.

```
N = 150;
B = 3;
F = [0 .25 .3 .4 .5 .6 .7 .75 1];
A = [1 1 0 0 0 0 0 1 1];
A1 = A(1:2);
A2 = A(3:7);
A3 = A(8:end);
F1 = F(1:2);
F2 = F(3:7);
F3 = F(8:end);
d = fdesign.arbmag('N,B,F,A',N,B,F1,A1,F2,A2,F3,A3);
Hd = design(d);
fvtool(Hd)
```

A response with two passbands — one roughly between 0 and 0.25 and the second between 0.75 and 1 — results from the mapping between F and A.
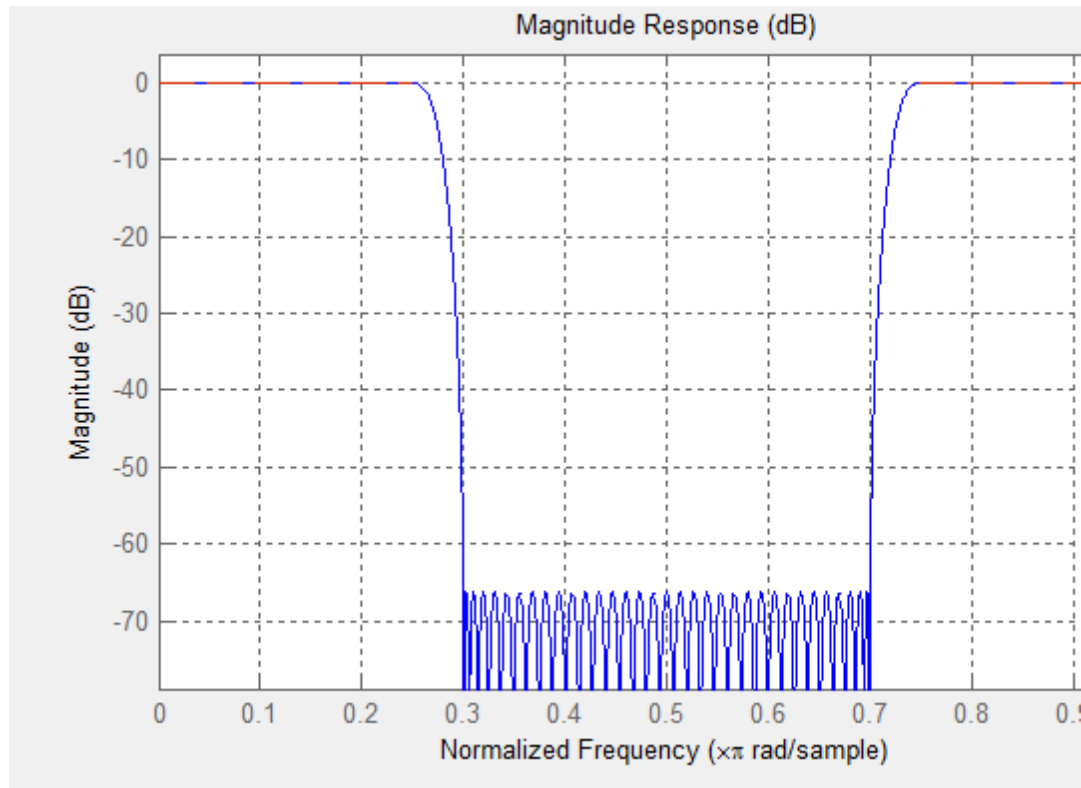
Design of a single band arbitrary-magnitude filter

Use fdesign.arbmag to design a single band equiripple filter.

```
n = 120;
f = linspace(0,1,100);  % 100 frequency points.
as = ones(1,100)-f*0.2;
absorb = [ones(1,30),(1-0.6*bohmanwin(10))',...
ones(1,5), (1-0.5*bohmanwin(8))',ones(1,47)];
a = as.*absorb;
d = fdesign.arbmag('N,F,A',n,f,a);
hd1 = design(d,'equiripple');
```

If you have the DSP System Toolbox, you can design a minimum-phase equiripple filter.

```
hd2 = design(d,'equiripple','MinPhase',true);
hfvt = fvtool([hd1 hd2],'analysis','polezero');
legend(hfvt,'Equiripple Filter','Minimum-phase Equiripple Filter');
```

Design of a multiband minimum order arbitrary-magnitude filter

Use fdesign.arbmag to design a multiband minimum order filter.

This example requires the DSP System Toolbox.

Place the notches at 0.25π and 0.55π radians/sample

```
d = fdesign.arbmag('B,F,A,R');
d.NBands = 5;
d.B1Frequencies = [0 0.2];
d.B1Amplitudes = [1 1];
d.B1Ripple = 0.25;
d.B2Frequencies = 0.25;
d.B2Amplitudes = 0;
d.B3Frequencies = [0.3 0.5];
d.B3Amplitudes = [1 1];
d.B3Ripple = 0.25;
d.B4Frequencies = 0.55;
d.B4Amplitudes = 0;
d.B5Frequencies = [0.6 1];
d.B5Amplitudes = [1 1];
d.B5Ripple = 0.25;
Hd = design(d,'equiripple');
fvtool(Hd)
```
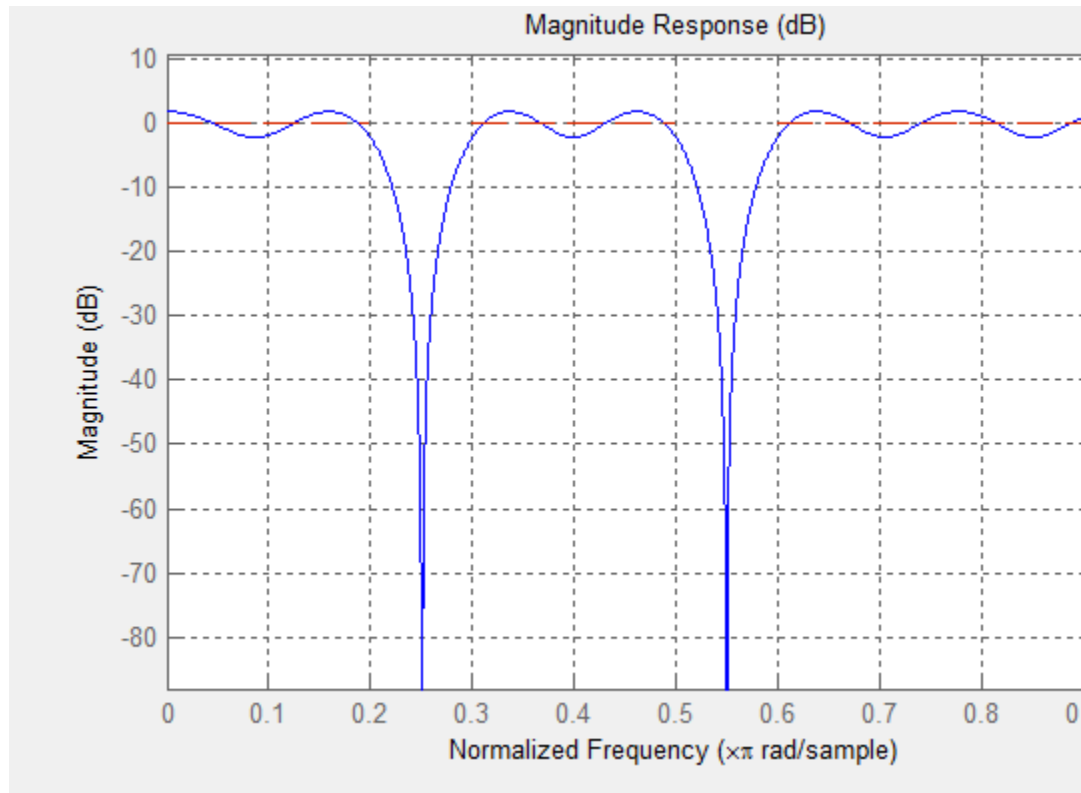
Design of a multiband constrained arbitrary-magnitude filter

Use fdesign.arbmag to design a multiband constrained FIR filter.

This example requires the DSP System Toolbox.

Force the frequency response at 0.15π radians/sample to 0 dB.

```
d = fdesign.arbmag('N,B,F,A,C',82,2);
d.B1Frequencies = [0 0.06 .1];
d.B1Amplitudes = [0 0 0];
d.B2Frequencies = [.15 1];
d.B2Amplitudes = [1 1];
```

```
% Design a filter with no constraints
Hd1 = design(d,'equiripple','B2ForcedFrequencyPoints',0.15);
% Add a constraint to the first band to increase attenuation
d.B1Constrained = true;
d.B1Ripple = .001;
Hd2 = design(d,'equiripple','B2ForcedFrequencyPoints',0.15);
hfvt = fvtool(Hd1,Hd2);
legend(hfvt,'Original Design','Design with Constrained Stopband Ripple
```
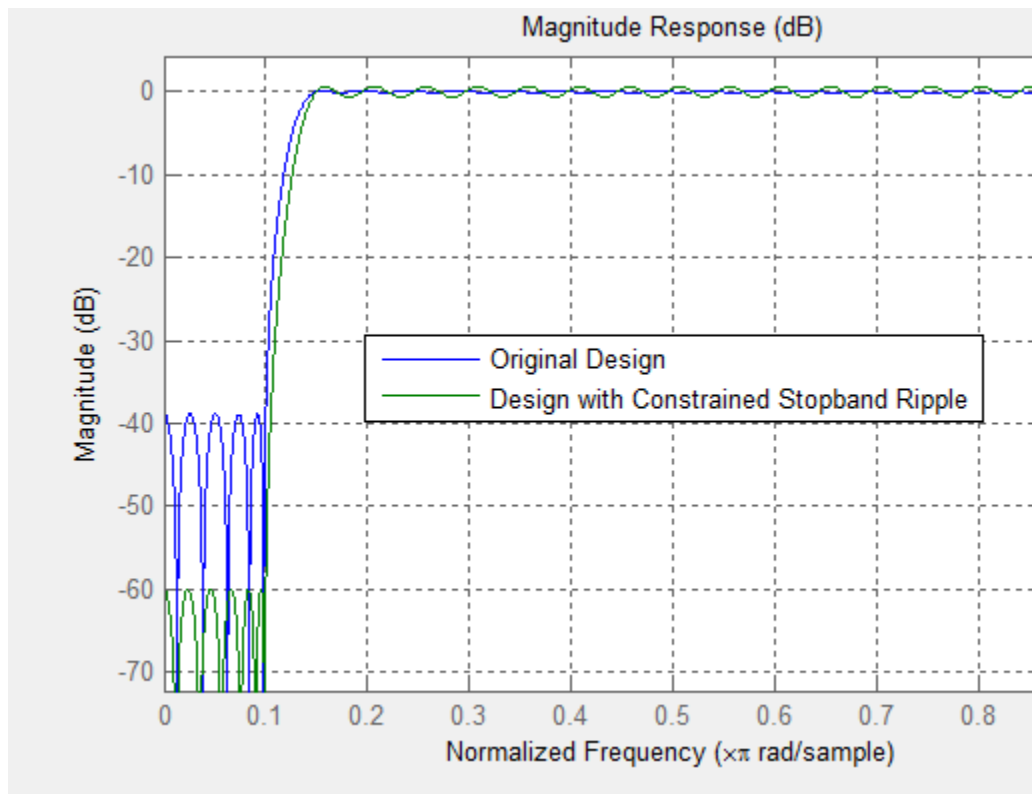

See Also   design | designmethods | fdesign

fdesign.arbmagnphase

Purpose   Arbitrary response magnitude and phase filter specification object

Syntax
```
d = fdesign.arbmagnphase
d = fdesign.arbmagnphase(specification)
d = fdesign.arbmagnphase(specification,specvalue1,specvalue2,
    ...)
d = fdesign.arbmagnphase(specvalue1,specvalue2,specvalue3)
d = fdesign.arbmagnphase(...,fs)
```

Description   `d = fdesign.arbmagnphase` constructs an arbitrary magnitude filter specification object `d`.

`d = fdesign.arbmagnphase(specification)` initializes the Specification property for specifications object `d` to the string in `specification`. The input argument `specification` must be one of the strings shown in the following table. Specification strings are not case sensitive.

Specification String	Description of Resulting Filter
n,f,h	Single band design (default). FIR and IIR (n is the order for both numerator and denominator).
n,b,f,h	FIR multiband design where b defines the number of bands.
nb,na,f,h	IIR single band design.

The following table describes the arguments in the strings.

Argument	Description
b	Number of bands in the multiband filter.
f	Frequency vector. Frequency values specified in f indicate locations where you provide specific filter response amplitudes. When you provide f you must also provide h which contains the response values.

Argument	Description
h	Complex frequency response values.
n	Filter order for FIR filters and the numerator and denominator orders for IIR filters (when not specified by nb and na).
nb	Numerator order for IIR filters.
na	Denominator order for IIR filter designs.

By default, this method assumes that all frequency specifications are supplied in normalized frequency.

Specifying f and h

f and h are the input arguments you use to define the filter response desired. Each frequency value you specify in f must have a corresponding response value in h. This example creates a filter with two passbands (b = 4) and shows how f and h are related. This example is for illustration only. It is not an actual filter.

Define the frequency vector f as [0 0.1 0.2 0.4 0.5 0.6 0.9 1.0]

Define the response vector h as [0 0.5 0.5 0.1 0.1 0.8 0.8 0]

These specifications connect f and h as shown in the following table.

f (Normalized Frequency)	h (Response Desired at f)
0	0
0.1	0.5
0.2	0.5
0.4	0.1
0.5	0.1
0.6	0.8

f (Normalized Frequency)	h (Response Desired at f)
0.9	0.8
1.0	0.0

A response with two passbands—one roughly between 0.1 and 0.2 and the second between 0.6 and 0.9—results from the mapping between f and h. Plotting f and h yields the following figure that resembles a filter with two passbands.
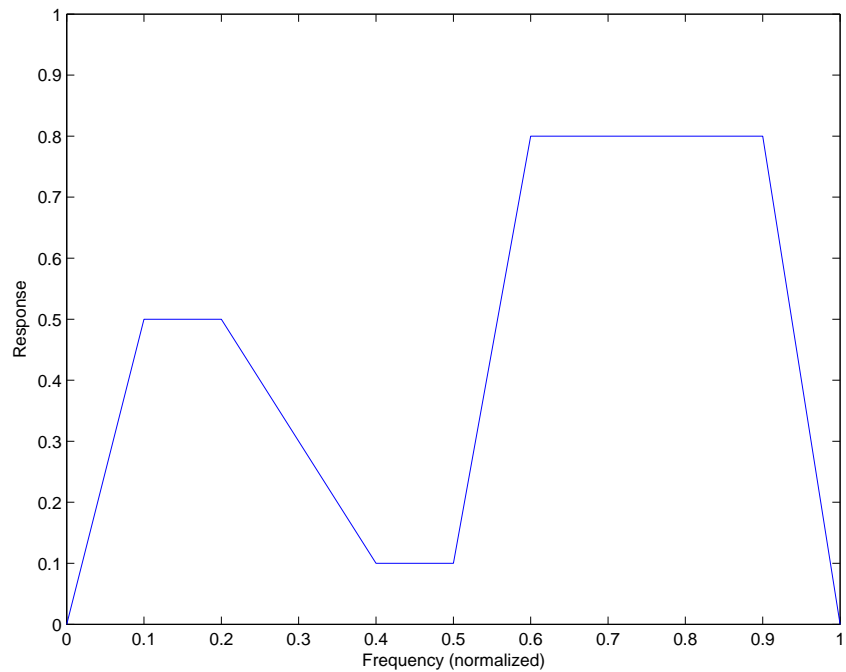
The second example in Examples shows this plot in more detail with a complex filter response for h. In the example, h uses complex values for the response.

Different specification types often have different design methods available. Use `designmethods(d)` to get a list of design methods available for a given specification string and specifications object.

`d = fdesign.arbmagnphase(specification,specvalue1,specvalue2,...)` initializes the filter specification object with `specvalue1`, `specvalue2`, and so on. Use `get(d,'description')` for descriptions of the various specifications `specvalue1`, `specvalue2`, ...`specn`.

`d = fdesign.arbmagnphase(specvalue1,specvalue2,specvalue3)` uses the default specification string `n,f,h`, setting the filter order, filter frequency vector, and the complex frequency response vector to the values `specvalue1`, `specvalue2`, and `specvalue3`.

`d = fdesign.arbmagnphase(...,fs)` specifies the sampling frequency in Hz. All other frequency specifications are also assumed to be in Hz when you specify `fs`.

Examples   Use `fdesign.arbmagnphase` to model a complex analog filter:

```
d=fdesign.arbmagnphase('n,f,h',100); % N=100, f and h set to defaults.
design(d,'freqsamp');
```

For a more complex example, design a bandpass filter with low group delay by specifying the desired delay and using f and h to define the filter bands.

```
n = 50;     % Group delay of a linear phase filter would be 25.
gd = 12;    % Set the desired group delay for the filter.
f1=linspace(0,.25,30); % Define the first stopband frequencies.
f2=linspace(.3,.56,40);% Define the passband frequencies.
f3=linspace(.62,1,30); % Define the second stopband frequencies.
h1 = zeros(size(f1)); % Specify the filter response at the freqs in f1.
h2 = exp(-1j*pi*gd*f2); % Specify the filter response at the freqs in f2.
h3 = zeros(size(f3)); % Specify the response at the freqs in f3.
d=fdesign.arbmagnphase('n,b,f,h',50,3,f1,h1,f2,h2,f3,h3);
D = design(d,'equiripple');
fvtool(D,'Analysis','freq');
```

fdesign.arbmagnphase
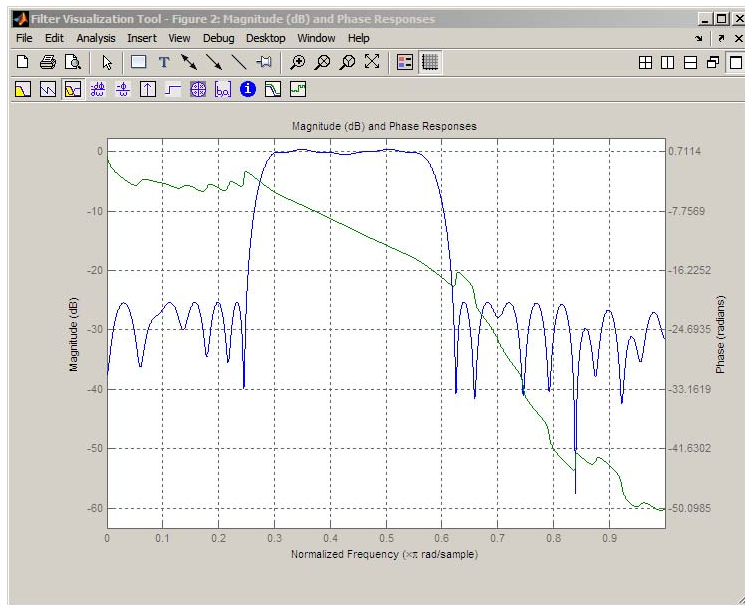
See Also   fdesign | design | designmethods | setspecs

Purpose   Audio weighting filter specification object

Syntax   *HAwf* = fdesign.audioweighting
                HAwf = fdesign.audioweighting(*spec*)
                HAwf = fdesign.audioweighting(*spec*,*specvalue1*,*specvalue2*)
                HAwf = fdesign.audioweighting(*specvalue1*,*specvalue2*)
                HAwf = fdesign.audioweighting(...,Fs)

Description Supported audio weighting filter types are: A weighting, C weighting,
                C-message, ITU-T 0.41, and ITU-R 468–4 weighting.

                HAwf = fdesign.audioweighting constructs an audio weighting filter
                specification object *HAwf* with a weighting type of A and a filter class
                of 1. Use the design method to instantiate a dfilt object based on
                the specifications in *HAwf*. Use designmethods to find valid filter
                design methods. Because the standards for audio weighting filters
                are in Hz, normalized frequency specifications are not supported for
                fdesign.audioweighting objects. The default sampling frequency for
                A weighting, C weighting, C-message, and ITU-T 0.41 filters is 48 kHz.
                The default sampling frequency for the ITU-R 468–4 filter is 80 kHz.
                If you invoke the normalizefreq method, a warning is issued when
                you instantiate the dfilt object and the default sampling frequencies
                in Hz are used.

                HAwf = fdesign.audioweighting(*spec*) returns an audio weighting
                filter specification object using default values for the specification string
                in *spec*. The following are valid entries for *spec*. The entries are not
                case sensitive.

                • 'WT,Class' (default *spec*)

                  The 'WT,Class' specification is valid for A weighting and C
                  weighting filters of class 1 or 2.

                  The weighting type is specified by the string: 'A' or 'C'. The class
                  is the scalar 1 or 2.

                  The default values for 'WT,Class' are 'A',1.

                • 'WT'

The `'WT'` specification is valid for C-message (default), ITU-T 0.41, and ITU-R 468–4 weighting filters.

The weighting type is specified by the string: `'Cmessage'`, `'ITUT041'`, or `'ITUR4684'`.

HAwf = fdesign.audioweighting(*spec*,*specvalue1*,*specvalue2*) constructs an audio weighting filter specification object *HAwf* and sets its specifications at construction time.

HAwf = fdesign.audioweighting(*specvalue1*,*specvalue2*) constructs an audio weighting filter specification object *HAwf* with the specification `'WT,Class'` using the values you provide. The valid weighting types are `'A'` or `'C'`.

HAwf = fdesign.audioweighting(...,Fs) specifies the sampling frequency in Hz. The sampling frequency is a scalar trailing all other input arguments.

Input Arguments

Parameter Name/Value Pairs

WT

Weighting type

The weighting type defines the frequency response of the filter. The valid weighting types are: A weighting, C weighting, C-message, ITU-T 0.41, and ITU-R 468–4 weighting. The weighting types are described in "Definitions" on page 7-513.

Class

Filter Class

Filter class is only applicable for A weighting and C weighting filters. The filter class describes the frequency-dependent tolerances specified in the relevant standards [1], [2]. There are two possible class values: 1 and 2. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in fvtool for the analysis of the filter design.

Default: 1

Definitions

A weighting

The specifications for the A weighting filter are found in ANSI standard S1.42-2001. The A weighting filter is based on the 40–phon Fletcher-Munson equal loudness contour [3]. One phon is equal to one dB sound pressure level (SPL) at one kHz. The Fletcher-Munson equal loudness contours are designed to account for frequency and level dependent differences in the perceived loudness of tonal stimuli. The 40–phon contour reflects the fact that the human auditory system is more sensitive to frequencies around 1–2 kHz than lower and higher frequencies. The filter roll off is more pronounced at lower frequencies and more modest at higher frequencies. While A weighting is based on the equal loudness contour for low-level (40–phon) tonal stimuli, it is commonly used in the United States for assessing potential health risks associated with noise exposure to narrowband and broadband stimuli at high levels.

C weighting

The specifications for the C weighting filter are found in ANSI standard S1.42-2001. The C weighting filter approximates the 100–phon Fletcher-Munson equal loudness contour for tonal stimuli. The C weighting magnitude response is essentially flat with 3–dB frequencies at 31.5 Hz and 8000 Hz. While C weighting is not as common as A weighting, sound level meters frequently have a C weighting filter option.

C-message

The specifications for the C–message weighting filter are found in Bell System Technical Reference, PUB 41009. C-message weighting filters are designed for measuring the impact of noise on telecommunications circuits used in speech transmission [6]. C-message weighting filters are commonly used in North America, while the ITU-T 0.41 filter is more commonly used outside of North America.

ITU-R 468–4

The specifications for the ITU-R 486–4 weighting filter are found in the International Telecommunication Union Recommendation ITU-R BS.468-4. ITU-R 486–4 is an equal loudness contour weighting filter. Unlike the A weighting filter, the ITU-R 468–4 filter describes subjective loudness judgements for broadband stimuli [4]. A common criticism of the A weighting filter is that it underestimates the loudness judgement of real-world stimuli particularly in the frequency band from about 1–9 kHz. A comparison of A weighting and ITU-R 468–4 weighting curves shows that the ITU-R 468–4 curve applies more gain between 1 and 10 kHz with a peak difference of approximately 12 dB around 6–7 kHz.

ITU-T 0.41

The specifications for the ITU-T 0.41 filter are found in the ITU-T Recommendation 0.41. ITU-T 0.41 weighting filters are designed for measuring the impact of noise on telecommunications circuits used in speech transmission [5]. ITU-T 0.41 weighting filters are commonly used outside of North America, while the C-message weighting filter is more common in North America.

Examples

Compare class 1 A weighting and ITU-R 468–4 filters between 0.1 and 12 kHz:

```
HawfA = fdesign.audioweighting('WT,Class','A',1,44.1e3);
% Sampling frequency is 44.1 kHz
HawfITUR = fdesign.audioweighting('WT','ITUR4684',44.1e3);
Afilter = design(HawfA);
ITURfilter = design(HawfITUR);
hfvt = fvtool([Afilter ITURfilter]);
axis([0.1 12 -80 20]);
legend(hfvt,'A-weighting','ITU-R 468-4');
```
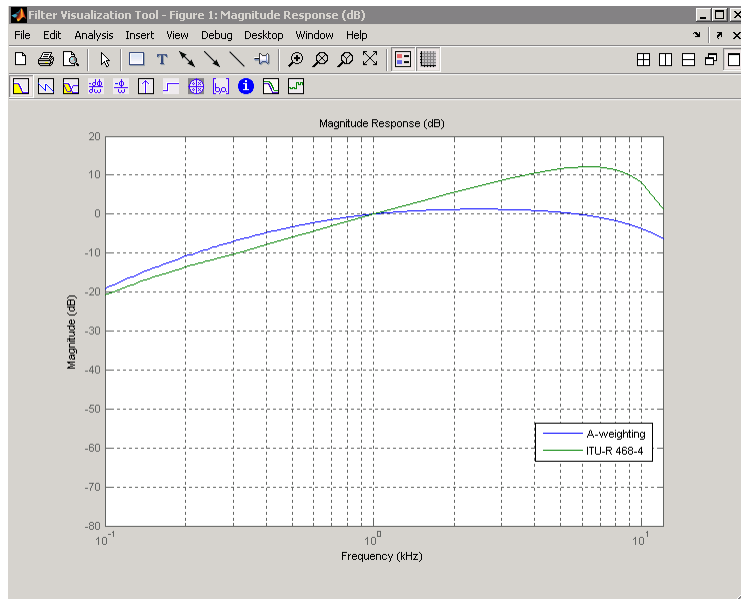
Compare C-message and ITU-T 0.41 filters:

```
hCmessage = fdesign.audioweighting('WT','Cmessage',24e3);
hITUT = fdesign.audioweighting('WT','ITUT041',24e3);
dCmessage = design(hCmessage);
dITUT = design(hITUT);
hfvt = fvtool([dCmessage dITUT]);
legend(hfvt,'C-Message Weighting','ITU-T 0.41 Weighting');
axis([0.1 10 -50 5]);
```
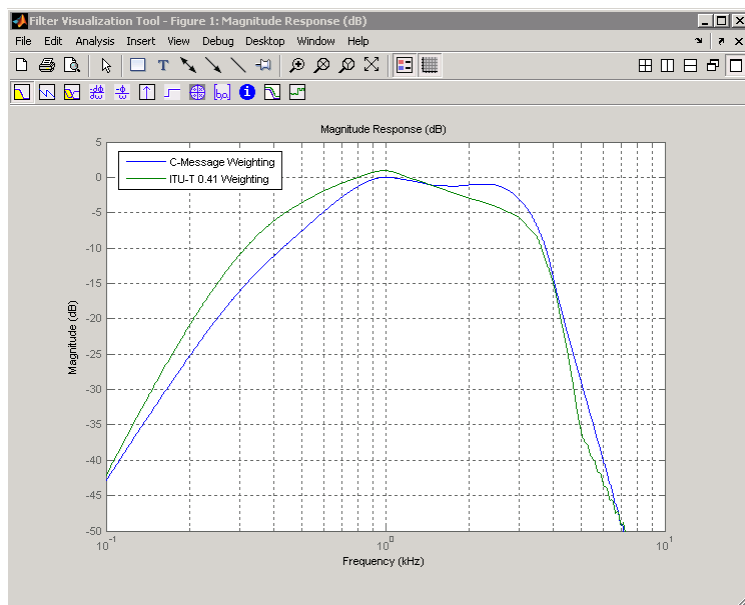
Construct an ITU-R 468–4 filter using all available design methods:

```
HAwf = fdesign.audioweighting('WT','ITUR4684');
ValidDesigns = designmethods(HAwf);
% returns iirlpnorm,equiripple,freqsamp in cell array
D = design(HAwf,'all'); % returns all designs
hfvt = fvtool(D);
legend(hfvt,'Least P-norm IIR','FIR Equiripple',...,
'FIR Frequency Sampling')
```
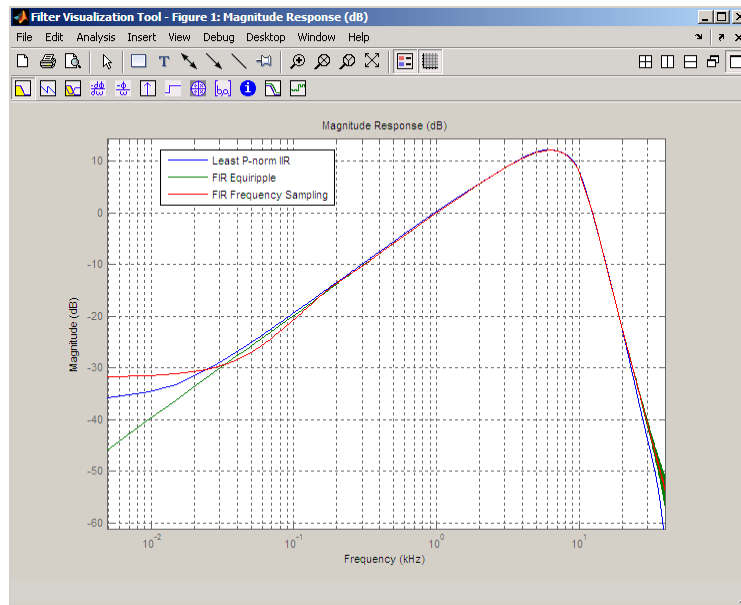
References

[1] *American National Standard Design Response of Weighting Networks for Acoustical Measurements*, ANSI S1.42-2001, Acoustical Society of America, New York, NY, 2001.

[2] *Electroacoustics Sound Level Meters Part 1: Specifications*, IEC 61672-1, First Edition 2002-05.

[3] Fletcher, H. and W.A. Munson. "Loudness, its definition, measurement and calculation." *Journal of the Acoustical Society of America*, Vol. 5, 1933, pp. 82–108.

[4] *Measurement of Audio-Frequency Noise Voltage Level in Sound Broadcasting*, International Telecommunication Union Recommendation ITU-R BS.468-4, 1986.

[5] *Psophometer for Use on Telephone-Type Circuits*, ITU-T Recommendation 0.41.

fdesign.audioweighting

[6] *Transmission Parameters Affecting Voiceband Data Transmission-Measuring Techniques*, Bell System Technical Reference, PUB 41009, 1972.

See Also　　design | designmethods | fdesign | fvtool

How To
- Audio Weighting Filters Example
- "Design a Filter in Fdesign — Process Overview"

Purpose   Bandpass filter specification object

Syntax
```
D = fdesign.bandpass
D = fdesign.bandpass(SPEC)
D = fdesign.bandpass(spec,specvalue1,specvalue2,...)
D = fdesign.bandpass(specvalue1,specvalue2,specvalue3,
specvalue4,...specvalue4,specvalue5,specvalue6)
D = fdesign.bandpass(...,Fs)
D = fdesign.bandpass(...,MAGUNITS)
```

Description   D = fdesign.bandpass constructs a bandpass filter specification object D, applying default values for the properties Fstop1, Fpass1, Fpass2, Fstop2, Astop1, Apass, and Astop2 — one possible set of values you use to specify a bandpass filter.

D = fdesign.bandpass(SPEC) constructs object D and sets its Specification property to SPEC. Entries in the SPEC string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for SPEC are shown below and used to define the bandpass filter. The strings are not case sensitive.

Note Specifications strings marked with an asterisk require the DSP System Toolbox software.

- 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2' (default spec)

- 'N,F3dB1,F3dB2'

- "N,F3dB1,F3dB2,Ap' *

- 'N,F3dB1,F3dB2,Ast' *

- 'N,F3dB1,F3dB2,Ast1,Ap,Ast2' *

- 'N,F3dB1,F3dB2,BWp *

- 'N,F3dB1,F3dB2,BWst' *

- 'N,Fc1,Fc2'

- `'N,Fc1,Fc2,Ast1,Ap,Ast2'`

- `'N,Fp1,Fp2,Ap'`

- `'N,Fp1,Fp2,Ast1,Ap,Ast2'`

- `'N,Fst1,Fp1,Fp2,Fst2'`

- `'N,Fst1,Fp1,Fp2,Fst2,C'` *

- `'N,Fst1,Fp1,Fp2,Fst2,Ap'` *

- `'N,Fst1,Fst2,Ast'`

- `'Nb,Na,Fst1,Fp1,Fp2,Fst2'` *

The string entries are defined as follows:

- `Ap` — amount of ripple allowed in the pass band. Also called Apass.

- `Ast1` — attenuation in the first stop band in decibels (the default units). Also called Astop1.

- `Ast2` — attenuation in the second stop band in decibels (the default units). Also called Astop2.

- `BWp` — bandwidth of the filter passband. Specified in normalized frequency units.

- `BWst` — bandwidth of the filter stopband. Specified in normalized frequency units.

- `C` — Constrained band flag. This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.

  In the specification string `'N,Fst1,Fp1,Fp2,Fst2,C'`, you cannot specify constraints in both stopbands and the passband simultaneously. You can specify constraints in any one or two bands.

- `F3dB1` — cutoff frequency for the point 3 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (IIR filters)

- F3dB2 — cutoff frequency for the point 3 dB point below the passband value for the second cutoff. Specified in normalized frequency units. (IIR filters)

- Fc1 — cutoff frequency for the point 6 dB point below the passband value for the first cutoff. Specified in normalized frequency units. (FIR filters)

- Fc2 — cutoff frequency for the point 6 dB point below the passband value for the second cutoff. Specified in normalized frequency units. (FIR filters)

- Fp1 — frequency at the edge of the start of the pass band. Specified in normalized frequency units. Also called Fpass1.

- Fp2 — frequency at the edge of the end of the pass band. Specified in normalized frequency units. Also called Fpass2.

- Fst1 — frequency at the edge of the start of the first stop band. Specified in normalized frequency units. Also called Fstop1.

- Fst2 — frequency at the edge of the start of the second stop band. Specified in normalized frequency units. Also called Fstop2.

- N — filter order for FIR filters. Or both the numerator and denominator orders for IIR filters when na and nb are not provided.

- Na — denominator order for IIR filters

- Nb — numerator order for IIR filters

Graphically, the filter specifications look similar to those shown in the following figure.

fdesign.bandpass
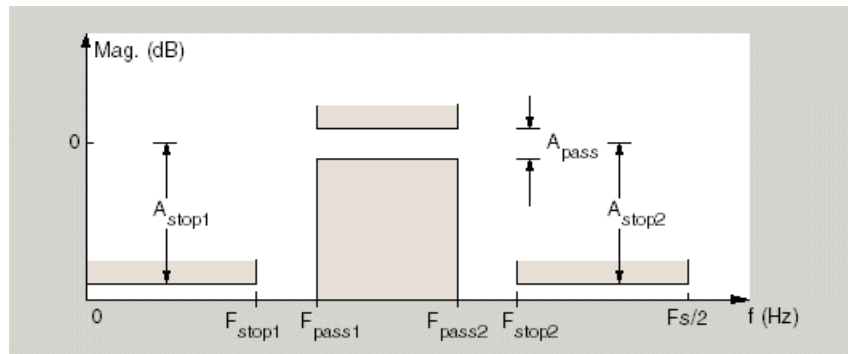
Regions between specification values like `Fst1` and `Fp1` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a bandpass filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design methods apply to an object and the `Specification` property value.

Use `designopts` to determine the design options for a given design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`D = fdesign.bandpass(spec,specvalue1,specvalue2,...)` constructs an object `D` and sets its specifications at construction time.

`D = fdesign.bandpass(specvalue1,specvalue2,specvalue3, specvalue4,...specvalue4,specvalue5,specvalue6)` constructs `D` with the default `Specification` property string, using the values you provide as input arguments for `specvalue1,specvalue2,specvalue3,specvalue4,specvalue4,specvalue5, specvalue6` and `specvalue7`.

`D = fdesign.bandpass(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

D = fdesign.bandpass(...,MAGUNITS) specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

- 'linear' — specify the magnitude in linear units
- 'dB' — specify the magnitude in dB (decibels)
- 'squared' — specify the magnitude in power units

When you omit the MAGUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   Filter a discrete-time signal with a bandpass filter. The signal is a sum of three discrete-time sinusoids, π/8, π/2, and 3π/4 radians/sample.

```
n = 0:159;
x = cos(pi/8*n)+cos(pi/2*n)+sin(3*pi/4*n);
```

Design an FIR equiripple bandpass filter to remove the lowest and highest discrete-time sinusoids.

```
d = fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2',1/4,3/8,5/8,6/8,
Hd = design(d,'equiripple');
```

Apply the filter to the discrete-time signal.

```
y = filter(Hd,x);
freq = 0:(2*pi)/length(x):pi;
xdft = fft(x);
ydft = fft(y);
plot(freq,abs(xdft(1:length(x)/2+1)));
hold on;
plot(freq,abs(ydft(1:length(x)/2+1)),'r','linewidth',2);
legend('Original Signal','Bandpass Signal');
```

fdesign.bandpass

Design an IIR Butterworth filter of order 10 with 3–dB frequencies of 1 and 1.2 kHz. The sampling frequency is 10 kHz

```
d = fdesign.bandpass('N,F3dB1,F3dB2',10,1e3,1.2e3,1e4);
Hd = design(d,'butter');
fvtool(Hd)
```

This example requires the DSP System Toolbox software.

Design a constrained-band FIR equiripple filter of order 100 with a passband of [1, 1.4] kHz. Both stopband attenuation values are constrained to 60 dB. The sampling frequency is 10 kHz.

```
d = fdesign.bandpass('N,Fst1,Fp1,Fp2,Fst2,C',100,800,1e3,1.4e3,1.6e3,1e4)
d.Stopband1Constrained = true; d.Astop1 = 60;
d.Stopband2Constrained = true; d.Astop2 = 60;
Hd = design(d,'equiripple');
fvtool(Hd);
measure(Hd)
```

The passband ripple is slightly over 2 dB. Because the design constrains both stopbands, you cannot constrain the passband ripple.

See Also   fdesign, fdesign.bandstop, fdesign.highpass, fdesign.lowpass

Purpose Bandstop filter specification object

Syntax
```
D = fdesign.bandstop
D = fdesign.bandstop(SPEC)
D = fdesign.bandstop(SPEC,specvalue1,specvalue2,...)
D = fdesign.bandstop(specvalue1,specvalue2,specvalue3,specvalue4,...
specvalue5,specvalue6,specvalue7)
D = fdesign.bandstop(...,Fs)
D = fdesign.bandstop(...,MAGUNITS)
```

Description D = fdesign.bandstop constructs a bandstop filter specification object D, applying default values for the properties Fpass1, Fstop1, Fstop2, Fpass2, Apass1, Astop1 and Apass2.

D = fdesign.bandstop(SPEC) constructs object D and sets the Specification property to SPEC. Entries in the SPEC string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for SPEC are shown below. The strings are not case sensitive.

Note Specifications strings marked with an asterisk require the DSP System Toolbox software.

- `'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2'` (default spec)

- `'N,F3dB1,F3dB2'`

- `'N,F3dB1,F3dB2,Ap'` *

- `'N,F3dB1,F3dB2,Ap,Ast'` *

- `'N,F3dB1,F3dB2,Ast'` *

- `'N,F3dB1,F3dB2,BWp'` *

- `'N,F3dB1,F3dB2,BWst'` *

- `'N,Fc1,Fc2'`

- `'N,Fc1,Fc2,Ap1,Ast,Ap2'`

- `'N,Fp1,Fp2,Ap'`

- `'N,Fp1,Fp2,Ap,Ast'`

- `'N,Fp1,Fst1,Fst2,Fp2'`

- `'N,Fp1,Fst1,Fst2,Fp2,C'` *

- `'N,Fp1,Fst1,Fst2,Fp2,Ap'` *

- `'N,Fst1,Fst2,Ast'`

- `'Nb,Na,Fp1,Fst1,Fst2,Fp2'` *

The string entries are defined as follows:

- `Ap` — amount of ripple allowed in the passband in decibels (the default units). Also called Apass.

- `Ap1` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass1.

- `Ap2` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass2.

- `Ast` — attenuation in the first stopband in decibels (the default units). Also called Astop1.

- `BWp` — bandwidth of the filter passband. Specified in normalized frequency units.

- `BWst` — bandwidth of the filter stopband. Specified in normalized frequency units.

- `C` — Constrained band flag. This enables you to specify passband ripple or stopband attenuation for fixed-order designs in one or two of the three bands.

  In the specification string `'N,Fp1,Fst1,Fst2,Fp2,C'`, you cannot specify constraints simultaneously in both passbands and the stopband. You can specify constraints in any one or two bands.

- F3dB1 — cutoff frequency for the point 3 dB point below the passband value for the first cutoff.

- F3dB2 — cutoff frequency for the point 3 dB point below the passband value for the second cutoff.

- Fc1 — cutoff frequency for the point 6 dB point below the passband value for the first cutoff. (FIR filters)

- Fc2 — cutoff frequency for the point 6 dB point below the passband value for the second cutoff. (FIR filters)

- Fp1 — frequency at the start of the pass band. Also called Fpass1.

- Fp2 — frequency at the end of the pass band. Also called Fpass2.

- Fst1 — frequency at the end of the first stop band. Also called Fstop1.

- Fst2 — frequency at the start of the second stop band. Also called Fstop2.

- N — filter order.

- Na — denominator order for IIR filters.

- Nb — numerator order for IIR filters.

Graphically, the filter specifications look similar to those shown in the following figure.
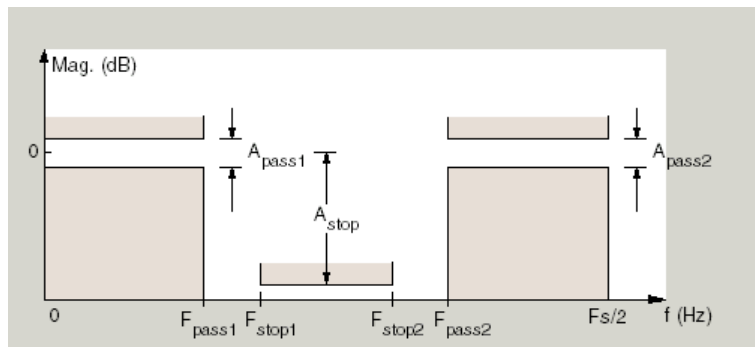
fdesign.bandstop

Regions between specification values like `Fp1` and `Fst1` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a bandstop filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design methods apply to an object and the `Specification` property value.

Use `designopts` to determine the design options for a given design method. Enter `help(D,METHOD)` at the MATLAB command line to obtain detailed help on the design options for a given design method, `METHOD`.

`D = fdesign.bandstop(SPEC,specvalue1,specvalue2,...)` constructs an object D and sets its specifications at construction time.

`D = fdesign.bandstop(specvalue1,specvalue2,specvalue3,specvalue4,...specvalue5,specvalue6,specvalue7)` constructs an object D with the default `Specification` property string , using the values you provide in `specvalue1,specvalue2,specvalue3,specvalue4,specvalue5,` `specvalue6` and `specvalue7`.

`D = fdesign.bandstop(...,Fs)` adds the argument Fs, specified in Hz to define the sampling frequency. If you specify the sampling frequency as a trailing scalar, all frequencies in the specifications are in Hz as well.

`D = fdesign.bandstop(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

- `'linear'` — specify the magnitude in linear units

- `'dB'` — specify the magnitude in dB (decibels)

- `'squared'` — specify the magnitude in power units

When you omit the MAGUNITS argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   Construct a bandstop filter to reject the discrete frequency band between
3π/8 and 5π/8 radians/sample. Apply the filter to a discrete-time signal
consisting of the superposition of three discrete-time sinusoids.

Design an FIR equiripple filter and view the magnitude response.

```
d = fdesign.bandstop('Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2',2/8,3/8,5/8,6/8,
Hd = design(d,'equiripple');
fvtool(Hd)
```

Construct the discrete-time signal to filter.

```
n = 0:99;
x = cos(pi/5*n)+sin(pi/2*n)+cos(4*pi/5*n);
y = filter(Hd,x);
xdft = fft(x);
ydft = fft(y);
freq = 0:(2*pi)/length(x):pi;
plot(freq,abs(xdft(1:length(x)/2+1)));
hold on;
plot(freq,abs(ydft(1:length(y)/2+1)),'r','linewidth',2);
xlabel('Radians/Sample'); ylabel('Magnitude');
legend('Original Signal','Bandstop Signal');
```

Create a Butterworth bandstop filter for data sampled at 10 kHz. The
stopband is [1,1.5] kHz. The order of the filter is 20.

```
d = fdesign.bandstop('N,F3dB1,F3dB2',20,1e3,1.5e3,1e4);
Hd = design(d,'butter');
fvtool(Hd);
```

Zoom in on the magnitude response plot to verify that the 3-dB down
points are located at 1 and 1.5 kHz.

The following example requires the DSP System Toolbox license.

Design a constrained-band FIR equiripple filter of order 100 for data sampled at 10 kHz. You can specify constraints on at most two of the three bands: two passbands and one stopband. In this example, you choose to constrain the passband ripple to be 0.5 dB in each passband. Design the filter, visualize the magnitude response and measure the filter's design.

```
d = fdesign.bandstop('N,Fp1,Fst1,Fst2,Fp2,C',100,800,1e3,1.5e3,1.7e3,1e4)
d.Passband1Constrained = true; d.Apass1 = 0.5;
d.Passband2Constrained = true; d.Apass2 = 0.5;
Hd = design(d,'equiripple');
fvtool(Hd);
measure(Hd)
```

With this order filter and passband ripple constraints, you achieve approximately 50 dB of stopband attentuation.

See Also   fdesign, fdesign.bandpass, fdesign.highpass, fdesign.lowpass

Purpose   CIC compensator filter specification object

Syntax
```
d= fdesign.ciccomp
d= fdesign.ciccomp(d,nsections,rcic)
d= fdesign.ciccomp(...,spec)
h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...)
```

Description   `d= fdesign.ciccomp` constructs a CIC compensator specifications object `d`, applying default values for the properties Fpass, Fstop, Apass, and Astop. In this syntax, the filter has two sections and the differential delay is 1.

Using `fdesign.ciccomp` with a design method creates a `dfilt` object, a single-rate discrete-time filter.

`d= fdesign.ciccomp(d,nsections,rcic)` constructs a CIC compensator specifications object with the filter differential delay set to `d`, the number of sections in the filter set to `nsections`, and the CIC rate change factor set to `rcic`. The default values of these parameters are: the differential delay equal to 1, the number of sections equal to 2, and the CIC rate change factor equal to 1.

If the CIC rate change factor is equal to 1, the filter passband response is an inverse sinc that is an approximation to the true inverse passband response of the CIC filter.

If you specify a CIC rate change factor not equal to 1, the filter passband response is an inverse Dirichlet sinc that matches exactly the true inverse passband response of the CIC filter.

`d= fdesign.ciccomp(...,spec)` constructs a CIC Compensator specifications object and sets its `Specification` property to *spec*. Entries in the spec string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for *spec* are shown in the list below. The strings are not case sensitive.

- `'fp,fst,ap,ast'` (default spec)

- `'n,fc,ap,ast'`

- `'n,fp,ap,ast'`

- `'n,fp,fst'`

- `'n,fst,ap,ast'`

The string entries are defined as follows:

- `ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.

- `ast` — attenuation in the stop band in decibels (the default units). Also called Astop.

- `fc` — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.

- `fp` — frequency at the end of the pass band. Specified in normalized frequency units. Also called Fpass.

- `fst` — frequency at the start of the stop band. Specified in normalized frequency units. Also called Fstop.

- `n` — filter order.

In graphic form, the filter specifications look like this:
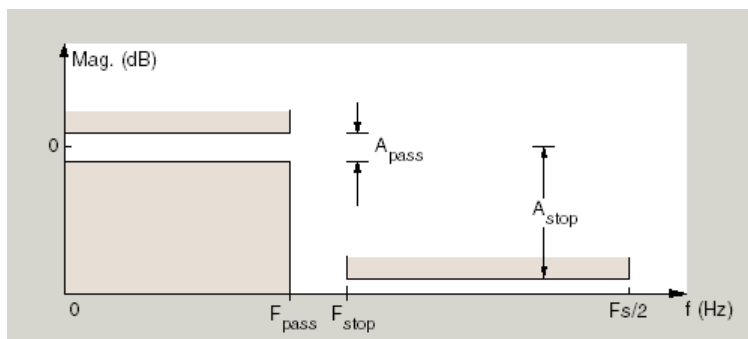
Regions between specification values like `fp` and `fst` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a CIC compensator specifications object change depending on the `Specification` string.

Use `designmethods` to determine which design method applies to an object and its specification string.

`h = fdesign.ciccomp(...,spec,specvalue1,specvalue2,...)` constructs an object and sets the specifications in the order they are specified in the `spec` input when you construct the object.

Designing CIC Compensators

Typically, when they develop filters, designers want flat passbands and transition regions that are as narrow as possible. CIC filters present a $(\sin x/x)$ profile in the passband and relatively wide transitions.

To compensate for this fall off in the passband, and to try to reduce the width of the transition region, you can use a CIC compensator filter that demonstrates an $(x/\sin x)$ profile in the passband. `fdesign.ciccomp` is specifically tailored to designing CIC compensators.

Here is a plot of a CIC filter and a compensator for that filter. The example that produces these filters follows the plot.

Magnitude Response (dB)
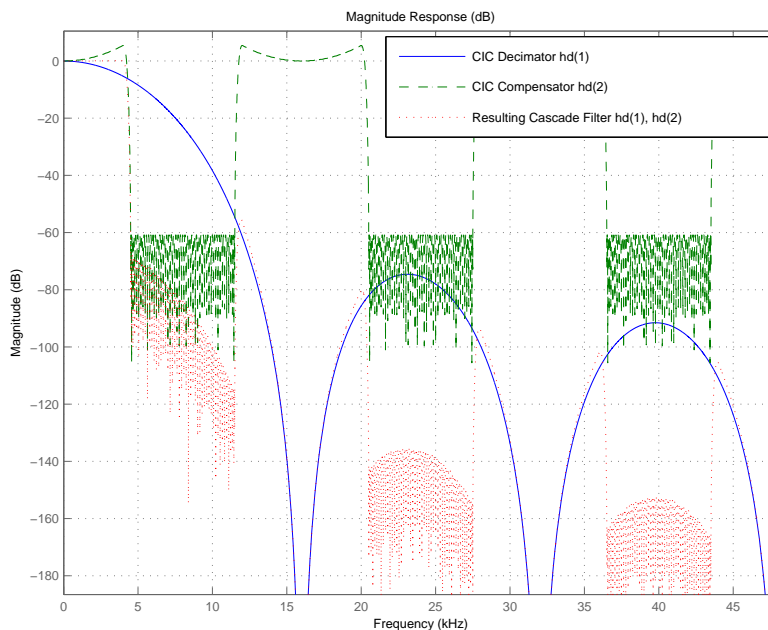
Given a CIC filter, how do you design a compensator for that filter? CIC compensators share three defining properties with the CIC filter — differential delay, d; number of sections, numberofsections; and the usable passband frequency, Fpass.

By taking the number of sections, passband, and differential delay from your CIC filter and using them in the definition of the CIC compensator, the resulting compensator filter effectively corrects for the passband droop of the CIC filter, and narrows the transition region.

As a demonstration of this concept, this example creates a CIC decimator and its compensator.

```
fs = 96e3;   % Input sampling frequency.
fpass = 4e3; % Frequency band of interest.
m = 6;   % Decimation factor.
hcic = design(fdesign.decimator(m,'cic',1,fpass,60,fs));
```

```
hd = cascade(dfilt.scalar(1/gain(hcic)),hcic);
hd(2) = design(fdesign.ciccomp(hcic.differentialdelay, ...
            hcic.numberofsections,fpass,4.5e3,.1,60,fs/m));
fvtool(hd(1),hd(2),...
cascade(hd(1),hd(2)),'Fs',[96e3 96e3/m 96e3])
```

You see the results in the preceding plot.

Examples   Designed to compensate for the rolloff inherent in CIC filters, CIC
compensators can improve the performance of your CIC design.
This example designs a compensator d with five sections and a
differential delay equal to one. The plot displayed after the code
shows the increasing gain in the passband that is characteristic of
CIC compensators, to overcome the droop in the CIC filter passband.
Ideally, cascading the CIC compensator with the CIC filter results in a
lowpass filter with flat passband response and narrow transition region.

```
h = fdesign.ciccomp;
set(h, 'NumberOfSections', 5, 'DifferentialDelay', 1);
hd = equiripple(h);
fvtool(hd);
```

Magnitude Response (dB)
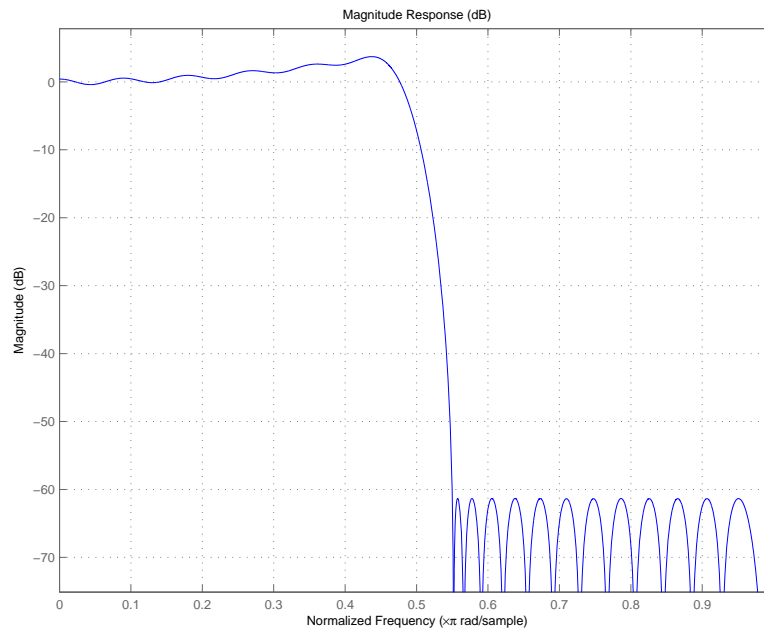
This compensator would work for a decimator or interpolator that had differential delay of 1 and 5 sections.

See Also   `fdesign.decimator` | `fdesign.interpolator`

Purpose   IIR comb filter specification object

Syntax
```
d=fdesign.comb
d=fdesign.comb(combtype)
d=fdesign(combtype,specstring)
d=fdesign(combtype,specstring,specvalue1,specvalue2,...)
d=fdesign.comb(...,Fs)
```

Description   fdesign.comb specifies a peaking or notching comb filter. Comb filters amplify or attenuate a set of harmonically related frequencies.

d=fdesign.comb creates a notching comb filter specification object and applies default values for the filter order (N=10) and quality factor (Q=16).

d=fdesign.comb(combtype) creates a comb filter specification object of the specified type and applies default values for the filter order and quality factor. The valid entries for combtype are shown in the following table. The entries are not case-sensitive.

Argument	Description
notch	creates a comb filter that attenuates a set of harmonically related frequencies.
peak	creates a comb filter that amplifies a set of harmonically related frequencies.

d=fdesign(combtype,*specstring*) creates a comb filter specification object of type combtype and sets its Specification property to *specstring* with default values. The entries in *specstring* determine the number of peaks or notches in the comb filter as well as their bandwidth and slope. Valid entries for *specstring* are shown below. The entries are not case-sensitive.

- 'N,Q' (default)

- ''N,BW'

- `'L,BW,GWB,Nsh'`

The following table describes the arguments in *specstring*.

Argument	Description
BW	Bandwidth of the notch or peak. By default the bandwidth is calculated at the point −3 dB down from the center frequency of the peak or notch. For example, setting BW=0.01 specifies that the −3 dB point will be +/- 0.005 (in normalized frequency) from the center of the notch or peak.
GWB	Gain at which the bandwidth is measured. This allows the user to specify the bandwidth of the notch or peak at a gain different from the −3 dB default.
L	Upsampling factor for a shelving filter of order Nsh. L determines the number of peaks or notches, which are equally spaced over the normalized frequency interval [-1,1].
N	Filter order. Specifies a filter with N+1 numerator and denominator coefficients.The filter will have N peaks or notches equally spaced over the interval [−1,1] .

Argument	Description
Nsh	Shelving filter order. Nsh represents a positive integer that determines the sharpness of the peaks or notches. The greater the value of the shelving filter order, the steeper the slope of the peak or notch. This results in a filter of order L*Nsh.
Q	Peak or notch quality factor. Q represents the ratio of the lowest center frequency peak or notch (not including DC) to the bandwidth calculated at the −3 dB point.

d=fdesign(combtype,*specstring*,specvalue1,specvalue2,...) creates an IIR comb filter specification object of type combtype and sets its Specification property to the values in specvalue1,specvalue2,...

d=fdesign.comb(...,Fs) creates an IIR comb filter specification object using the sampling frequency, Fs, of the signal to be filtered. The function assumes that Fs is in Hertz and must be specified as a scalar trailing all other provided values.

Examples These examples demonstrate how to create IIR comb filter specification objects. First, create a default specification object.

```
d=fdesign.comb;
```

In the next example, create a notching filter of order 8 with a bandwidth of 0.02 (normalized frequency) referenced to the −3 dB point.

```
d  = fdesign.comb('notch','N,BW',8,0.02);
Hd = design(d);
fvtool(Hd)
```
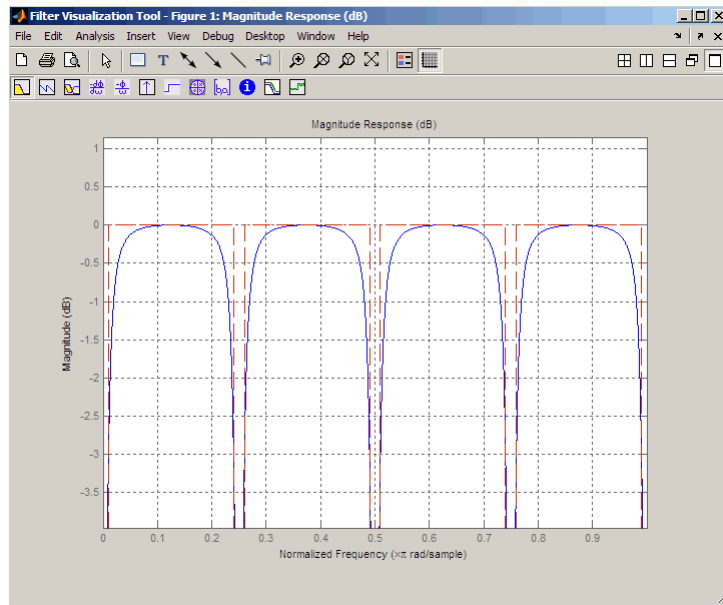
Next, create a peaking comb filter with 5 peaks and a quality factor of 25.

```
d = fdesign.comb('peak','N,Q',5,25);
Hd = design(d);
fvtool(Hd)
```
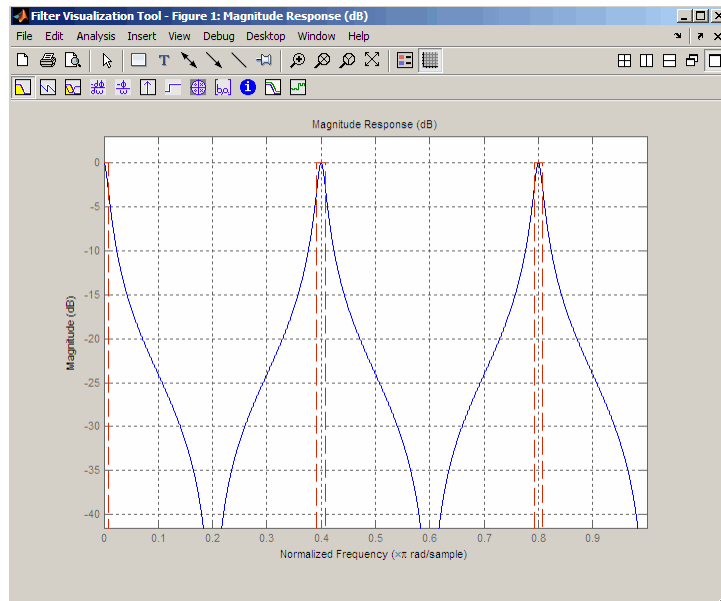
In the next example, create a notching filter to remove interference at 60 Hz and its harmonics. The following code creates a filter with 10 notches and a notch bandwidth of 5 Hz referenced to the -4dB level. The filter has a shelving filter order of 4 and a sampling frequency of 600 Hz. Because the notches are equidistantly spaced in the interval [-300, 300] Hz, they occur at multiples of 60 Hz.

```
d = fdesign.comb('notch','L,BW,GBW,Nsh',10,5,-4,4,600);
Hd=design(d);
fvtool(Hd)
```
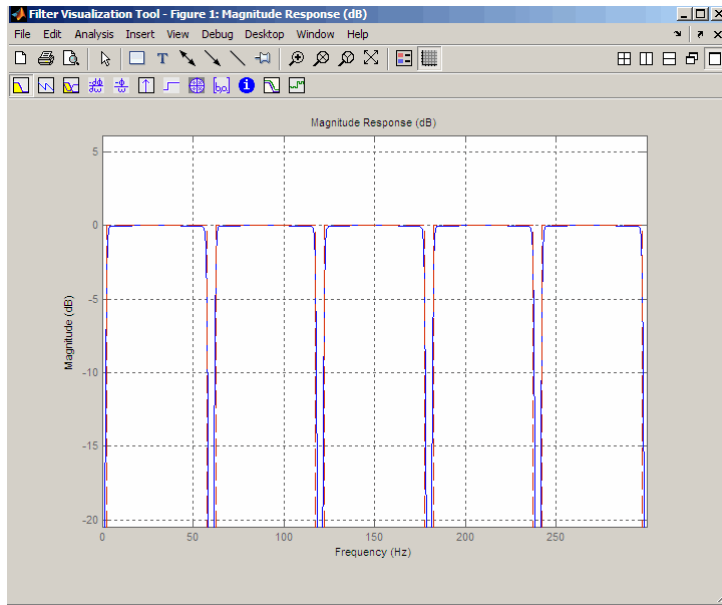
Purpose   Decimator filter specification object

Syntax
```
D = fdesign.decimator(M)
D = fdesign.decimator(M,RESPONSE)
D = fdesign.decimator(M,PULSESHAPINGRESPONSE,SPS)
D = fdesign.decimator(M,CICRESPONSE,D)
D = fdesign.decimator(M,RESPONSE,SPEC)
D = fdesign.decimator(...,SPEC,specvalue1,specvalue2,...)
D = fdesign.decimator(...,Fs)
D = fdesign.decimator(...,MAGUNITS)
```

Description   `D = fdesign.decimator(M)` constructs a decimator filter specification object D with the DecimationFactor property equal to the positive integer M and the Response property set to `'Nyquist'`. The default values for the transition width and stopband attenuation in the Nyquist design are 0.1π radians/sample and 80 dB. If M is unspecified, M defaults to 2.

`D = fdesign.decimator(M,RESPONSE)` constructs a decimator specification object with the decimation factor M and the `'Response'` property.

`D = fdesign.decimator(M,PULSESHAPINGRESPONSE,SPS)` constructs a pulse-shaping decimator specification object with the decimation factor M and `'Response'` property equal to one of the supported pulse-shaping filters: `'Gaussian'`, `'Raised Cosine'`, or `'Square Root Raised Cosine'`. SPS is the samples per symbol. The samples per symbol, SPS, must precede any specification string.

`D = fdesign.decimator(M,CICRESPONSE,D)` constructs a CIC or CIC compensator decimator specification object with the decimation factor, M, `'Response'` property equal to `'CIC'` or `'CICCOMP'`, and D equal to the differential delay. The differential delay, D, must precede any specification string.

Because you are designing multirate filters, the specification strings available are not the same as the specifications for designing single-rate filters. The decimation factor M is not included in the specification strings. Different filter responses support different specifications. The

following table lists the supported response types and specification strings. The strings are not case sensitive.

Design String	Valid Specification Strings
'Arbitrary Magnitude'	See `fdesign.arbmag` for a description of the specification string entries. • `'N,F,A'` (default string) • `'N,B,F,A'`
'Arbitrary Magnitude and Phase'	See `fdesign.arbmagnphase` for a description of the specification string entries. • `'N,F,H'` (default string) • `'N,B,F,H'`
'Bandpass'	See `fdesign.bandpass` for a description of the specification string entries. • `'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'` (default string) • `'N,Fc1,Fc2'` • `'N,Fst1,Fp1,Fp2,Fst2'`
'Bandstop'	See `fdesign.bandstop` for a description of the specification string entries.
'CIC'	`'Fp,Ast'` — Only valid specification. `Fp` is the passband frequency and `Ast` is the stopband attenuation in decibels. To specify a CIC decimator, include the differential delay after `'CIC'` and before the filter specification string: `'Fp,Ast'`. For example: `d = fdesign.decimator(2,'cic',4,'Fp,Ast',0.4,40);`

Design String	Valid Specification Strings
`'CIC Compensator'`	See `fdesign.ciccomp` for a description of the specification string entries. • `'Fp,Fst,Ap,Ast'` (default string) • `'N,Fc,Ap,Ast'` • `'N,Fp,Ap,Ast'` • `'N,Fp,Fst'` • `'N,Fst,Ap,Ast'` To specify a CIC compensator decimator, include the differential delay after `'CICCOMP'` and before the filter specification string.. For example: `d = fdesign.decimator(2,'ciccomp',4);`
`'Differentiator'`	`'N'` — filter order
`'Gaussian'`	`'Nsym,BT` — Nsym is the filter order in symbols and BT is the bandwidth-symbol time product. The specification string must be preceded by an integer-valued `SamplesPerSymbol`.
`'Halfband`	See `fdesign.halfband` for a description of the specification string entries. • `'TW,Ast'` (default string) • `'N,TW'` • `` `N' `` • `'N,Ast'` If you use the quasi-linear IIR design method, `iirlinphase`, with a halfband specification, the interpolation factor must be 2.

Design String	Valid Specification Strings
'Highpass'	See fdesign.highpass for a description of the specification string entries. • 'Fst,Fp,Ast,Ap' (default string) • 'N,F3db' • 'N,Fc' • 'N,Fc,Ast,Ap' • 'N,Fp,Ast,Ap' • 'N,Fst,Ast,Ap' • 'N,Fst,Fp' • 'N,Fst,Ast,Ap' • 'N,Fst,Fp,Ast'
'Hilbert'	See fdesign.hilbert for a description of the specification string entries. • 'N,TW' (default string) • 'TW,Ap'
'Inverse-sinc Lowpass'	See fdesign.isinclp for a description of the specification string entries. • 'Fp,Fst,Ap,Ast' (default string)) • 'N,Fc,Ap,Ast' • 'N,Fp,Fst' • 'N,Fst,Ap,Ast'

Design String	Valid Specification Strings
`'Inverse-sinc Highpass'`	See `fdesign.isinchp` for a description of the specification string entries. • `'Fst,Fp,Ast,Ap'` (default string)) • `'N,Fc,Ast,Ap'` • `'N,Fst,Fp'` • `'N,Fst,Ast,Ap'`
`'Lowpass'`	See `fdesign.lowpass` for a description of the specification string entries. • `'Fp,Fst,Ap,Ast'` (default string) • `'N,F3dB'` • `'N,Fc'` • `'N,Fc,Ap,Ast'` • `'N,Fp,Ap,Ast'` • `'N,Fp,Fst'` • `'N,Fp,Fst,Ap'` • `'N,Fp,Fst,Ast'` • `'N,Fst,Ap,Ast'`

Design String	Valid Specification Strings
`'Nyquist'`	See `fdesign.nyquist` for a description of the specification string entries. For all Nyquist specifications, you must specify the *L*th band. This typically corresponds to the `DecimationFactor`. • `'TW,Ast'` (default string) • `'N'` • `'N,Ast'` • `'N,Ast'`
`'Raised Cosine'`	See `fdesign.pulseshaping` for a description of the specification string entries. • `'Ast,Beta'` (default string) • `'Nsym,Beta'` • `'N,Beta'` The specification string must be preceded by an integer-valued `SamplesPerSymbol`.
`'Square Root Raised Cosine'`	See `fdesign.pulseshaping` for a description of the specification string entries. • `'Ast,Beta'` (default string) • `'Nsym,Beta'` • `'N,Beta'` The specification string must be preceded by an integer-valued `SamplesPerSymbol`.

`D = fdesign.decimator(M,RESPONSE,SPEC)` constructs object `D` and sets the `Specification` property to `SPEC` for the response type, `RESPONSE`. Entries in the `SPEC` string represent various filter response features,

such as the filter order, that govern the filter design. Valid entries for SPEC depend on the RESPONSE type.

Because you are designing multirate filters, the specification strings available are not the same as the specifications for designing single-rate filters with such response types as fdesign.lowpass. The strings are not case sensitive.

The decimation factor M is not in the specification strings.

D = fdesign.decimator(...,SPEC,specvalue1,specvalue2,...) constructs an object D and sets its specifications at construction time.

D = fdesign.decimator(...,Fs) provides the sampling frequency of the signal to be filtered. Fs must be specified as a scalar trailing the other numerical values provided. Fs is assumed to be in Hz as are all other frequency values provided.

D = fdesign.decimator(...,MAGUNITS) specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

- 'linear' — specify the magnitude in linear units.

- 'dB' — specify the magnitude in dB (decibels).

- 'squared' — specify the magnitude in power units.

When you omit the MAGUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   These examples show how to construct decimating filter specification objects. First, create a default specifications object without using input arguments except for the decimation factor m.

```
d = fdesign.decimator(2,'Nyquist',2,0.1,80) % Set tw=0.1, and ast=80.
```

Now create an object by passing a specification type string 'fst1,fp1,fp2,fst2,ast1,ap,ast2' and a design — the resulting object uses default values for the filter specifications. You must provide

the design input argument, bandpass in this example, when you include a specification.

```
d=fdesign.decimator(8,'bandpass',...
'fst1,fp1,fp2,fst2,ast1,ap,ast2');
```

Create another decimating filter specification object, passing the specification values to the object rather than accepting the default values for fp,fst,ap,ast.

```
d=fdesign.decimator(3,'lowpass',.45,0.55,.1,60);
```

Now pass the filter specifications that correspond to the specifications — n,fc,ap,ast.

```
d=fdesign.decimator(3,'ciccomp',1,2,'n,fc,ap,ast',...
20,0.45,.05,50);
```

Now design a decimator using the equiripple design method.

```
hm = equiripple(d);
```

Pass a new specification type for the filter, specifying the filter order. Note that the inputs must include the differential delay dd with the CIC input argument to design a CIC specification object.

```
m = 5;
dd = 2;
d = fdesign.decimator(m,'cic',dd,'fp,ast',0.55,55);
```

In this example, you specify a sampling frequency as the last input argument. Here is it 1000 Hz. Design an equiripple filter and plot the magnitude response:

```
d=fdesign.decimator(8,'bandpass','fst1,fp1,fp2,fst2,ast1,ap,ast2',...
100,150,250,300,50,.05,50,1000);
fvtool(design(d,'equiripple'))
```
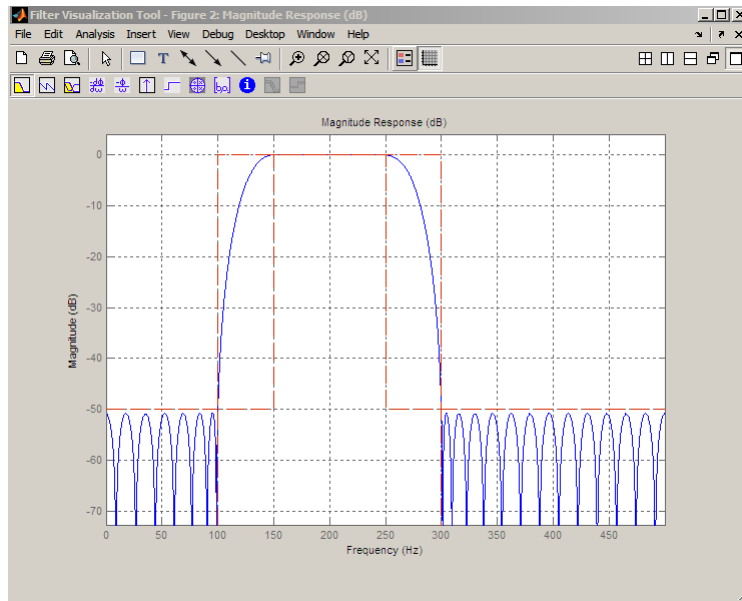
See Also

fdesign | fdesign.arbmag | fdesign.arbmagnphase | fdesign.interpolator | fdesign.rsrc

fdesign.differentiator

Purpose   Differentiator filter specification object

Syntax   D = fdesign.differentiator
                   D = fdesign.differentiator(SPEC)
                   D = fdesign.differentiator(SPEC,specvalue1,specvalue2, ...)
                   D = fdesign.differentiator(specvalue1)
                   D = fdesign.differentiator(...,Fs)
                   D = fdesign.differentiator(...,MAGUNITS)

Description   D = fdesign.differentiator constructs a default differentiator filter
                   designer D with the filter order set to 31.

                   D = fdesign.differentiator(SPEC) initializes the filter designer
                   Specification property to SPEC. You provide one of the following
                   strings as input to replace SPEC. The string you provide is not case
                   sensitive.

                   Note Specifications strings marked with an asterisk require the DSP
                   System Toolbox software.

                   - 'N' — Full band differentiator (default)

                   - 'N,Fp,Fst' — Partial band differentiator

                   - 'N,Fp,Fst,Ap' — Partial band differentiator *

                   - 'N,Fp,Fst,Ast' — Partial band differentiator *

                   - 'Ap' — Minimum order full band differentiator *

                   - 'Fp,Fst,Ap,Ast' — Minimum order partial band differentiator *

                   The string entries are defined as follows:

                   - Ap — amount of ripple allowed in the pass band in decibels (the
                     default units). Also called Apass.

                   - Ast — attenuation in the stop band in decibels (the default units).
                     Also called Astop.

- Fp — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.

- Fst — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.

- N — filter order.

By default, fdesign.differentiator assumes that all frequency specifications are provided in normalized frequency units. Also, decibels is the default for all magnitude specifications.

Use designopts to determine the design options for a given design method. Enter help(D,METHOD) at the MATLAB command line to obtain detailed help on the design options for a given design method, METHOD.

D = fdesign.differentiator(SPEC,specvalue1,specvalue2, ...) initializes the filter designer specifications in SPEC with specvalue1, specvalue2, and so on. To get a description of the specifications specvalue1, specvalue2, and more, enter

```
get(d,'description')
```

at the Command prompt.

D = fdesign.differentiator(specvalue1) assumes the default specification string N, setting the filter order to the value you provide.

D = fdesign.differentiator(...,Fs) adds the argument Fs, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

D = fdesign.differentiator(...,MAGUNITS) specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

- 'linear' — specify the magnitude in linear units

- 'dB' — specify the magnitude in dB (decibels)

- 'squared' — specify the magnitude in power units

fdesign.differentiator

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples

Use an FIR equiripple differentiator to transform frequency modulation into amplitude modulation, which can be detected using an envelope detector.

Modulate a message signal consisting of a 20-Hz sine wave with a 1 kHz carrier frequency. The sampling frequency is 10 kHz .

```
t = linspace(0,1,1e4);
x = cos(2*pi*20*t);
Fc = 1e3;
Fs = 1e4;
y = modulate(x,Fc,Fs,'fm');
```

Design the equiripple FIR differentiator of order 31.

```
d = fdesign.differentiator(31,1e4);
Hd = design(d,'equiripple');
```

Filter the modulated signal and take the Hilbert transform to obtain the envelope.

```
y1 = filter(Hd,y);
y1 = hilbert(y1);
% Plot the envelope
plot(t.*1000,abs(y1));
xlabel('Milliseconds'); ylabel('Magnitude');
grid on;
title('Envelope of the Demodulated Signal');
```
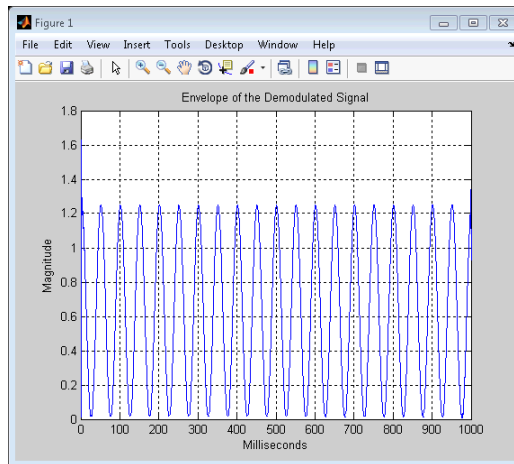
From the preceding figure, you see that the envelope completes two cycles every 100 milliseconds. The envelope is oscillating at 20 Hz, which corresponds to the frequency of the message signal.

Design an FIR differentiator using least squares and plot the zero phase response.

```
d = fdesign.differentiator(33); % Filter order is 33.
hd = design(d,'firls');
fvtool(hd,'magnitudedisplay','zero-phase',...
'frequencyrange','[-pi, pi)')
```
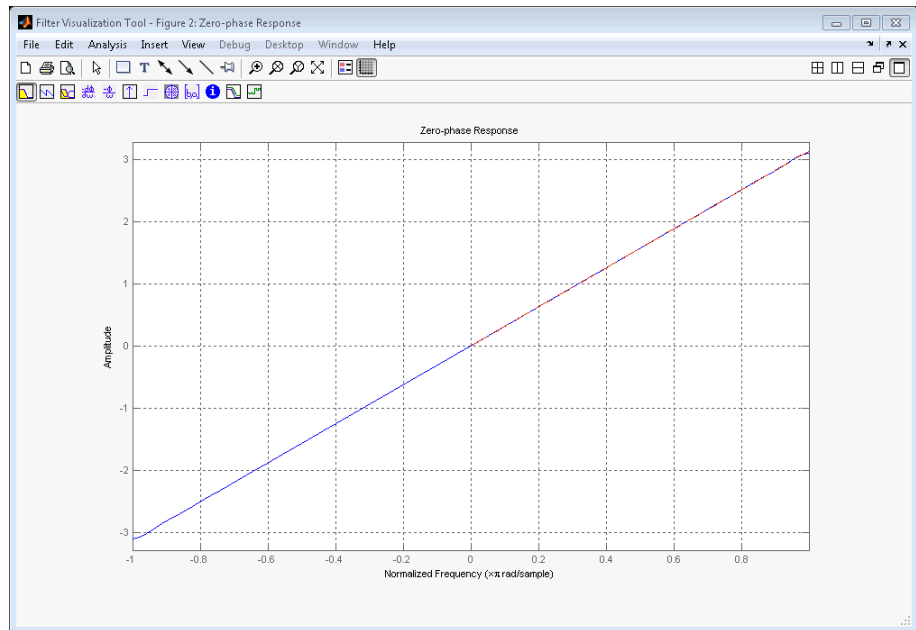
Design a narrow band differentiator. Differentiate the first 25 percent of the frequencies in the Nyquist range and filter the higher frequencies.

```
Fs=20000; %sampling frequency
d = fdesign.differentiator('N,Fp,Fst',54,2500,3000,Fs);
Hd= design(d,'equiripple');
% Weight the stopband to increase attenuation
Hd1 = design(d,'equiripple','Wstop',4);
hfvt = fvtool(Hd,Hd1,'magnitudedisplay','zero-phase',...
'frequencyrange','[O, Fs/2)');
legend(hfvt,'Without stopband weighting',...
'With stopband weighting');
```
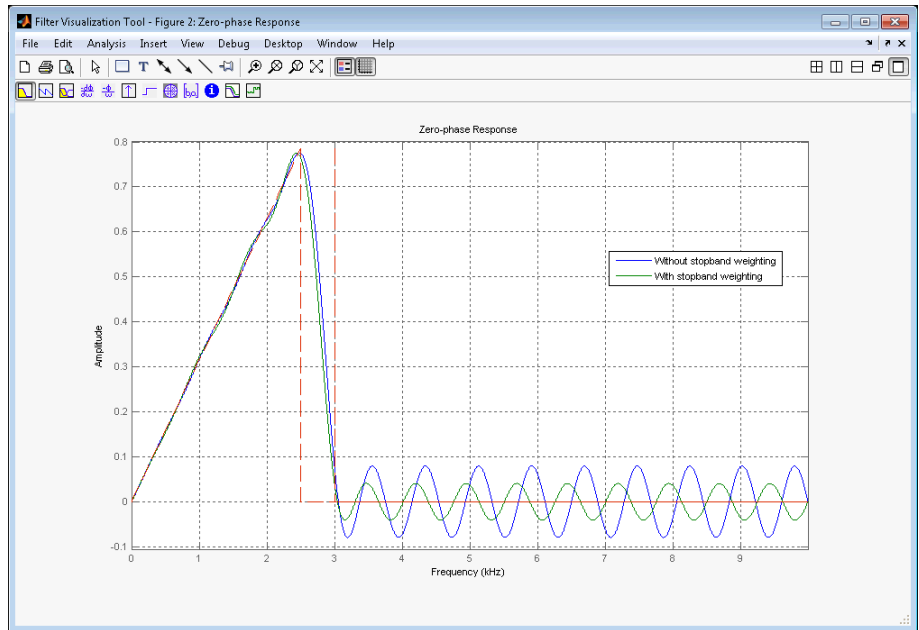
See Also   design | fdesign

fdesign.fracdelay

Purpose   Fractional delay filter specification object

Syntax
```
d = fdesign.fracdelay(delta)
d = fdesign.fracdelay(delta,'N')
d = fdesign.fracdelay(delta,'N',n)
d = fdesign.fracdelay(delta,n)
d = fdesign.fracdelay(...,fs)
```

Description   `d = fdesign.fracdelay(delta)` constructs a default fractional delay filter designer `d` with the filter order set to 3 and the delay value set to `delta`. The fractional delay `delta` must be between 0 and 1 samples.

`d = fdesign.fracdelay(delta,'N')` initializes the filter designer specification string to `N`, where `N` specifies the fractional delay filter order and defaults to filter order of 3.

Use `designmethods(d)` to get a list of the design methods available for a specification string.

`d = fdesign.fracdelay(delta,'N',n)` initializes the filter designer to specification string `N` and sets the filter order to `n`.

`d = fdesign.fracdelay(delta,n)` assumes the default specification `N`, filter order, and sets the filter order to the value you provide in input `n`.

`d = fdesign.fracdelay(...,fs)` adds the argument `fs`, specified in units of Hertz (Hz) to define the sampling frequency. In this case, specify the fractional delay `delta` to be between 0 and 1/`fs`.

Examples   Design a second-order fractional delay filter of 0.2 samples using the Lagrange method. Implement the filter using a Farrow fractional delay (`fd`) structure.

```
d = fdesign.fracdelay(0.2,'N',2);
hd = design(d,'lagrange','filterstructure','farrowfd');
fvtool(hd,'analysis','grpdelay')
```

Design a cubic fractional delay filter with a sampling frequency of 8 kHz and fractional delay of 50 microseconds using the Lagrange method.

```
d = fdesign.fracdelay(50e-6,'N',3,8000);
Hd = design(d, 'lagrange', 'FilterStructure', 'farrowfd');
```

See Also   design | designopts | fdesign | setspecs

fdesign.halfband

Purpose　Halfband filter specification object

Syntax
```
d = fdesign.halfband
d = fdesign.halfband('type',type)
d = fdesign.halfband(spec)
d = fdesign.halfband(spec,specvalue1,specvalue2,...)
d = fdesign.halfband(specvalue1,specvalue2)
d = fdesign.halfband(...,fs)
d = fdesign.halfband(...,magunits)
```

Description　`d = fdesign.halfband` constructs a halfband filter specification object `d`, applying default values for the properties `tw` and `ast`.

Using `fdesign.halfband` with a design method generates a `dfilt` object.

`d = fdesign.halfband('type',type)` initializes the filter designer 'Type' property with `type`. "type" must be either `lowpass` or `highpass` and is not case sensitive.

`d = fdesign.halfband(spec)` constructs object `d` and sets its 'Specification' to *spec*. Entries in the `spec` string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for *spec* are shown below. The strings are not case sensitive.

- tw,ast (default `spec`)
- n,tw
- n
- n,ast

The string entries are defined as follows:

- ast — attenuation in the stop band in decibels (the default units).
- n — filter order.
- tw — width of the transition region between the pass and stop bands. Specified in normalized frequency units.

By default, all frequency specifications are assumed to be in normalized frequency units. Moreover, all magnitude specifications are assumed to be in dB. Different specification types may have different design methods available.

The filter design methods that apply to a halfband filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design method applies to an object and its specification string. Different filter design methods also have options that you can specify. Use `designopts` with the design method string to see the available options. For example:

```
f=fdesign.halfband('N,TW');
designmethods(f)
```

`d = fdesign.halfband(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.halfband(specvalue1,specvalue2)` constructs an object `d` assuming the default `Specification` property string `tw,ast`, using the values you provide for the input arguments `specvalue1` and `specvalue2` for `tw` and `ast`.

`d = fdesign.halfband(...,fs)` adds the argument `fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.halfband(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. `magunits` can be one of

- `linear` — specify the magnitude in linear units

- `dB` — specify the magnitude in dB (decibels)

- `squared` — specify the magnitude in power units

When you omit the `magunits` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

fdesign.halfband

Examples

Create a default halfband filter specifications object:

```
d=fdesign.halfband;
```

Create another halfband filter object, passing the specification values to the object rather than accepting the default values for n and ast.

```
 d = fdesign.halfband('n,ast', 42, 80);
```

For another example, pass the filter values that correspond to the default Specification — n,ast.

```
d = fdesign.halfband(.01, 80);
```

This example designs an equiripple FIR filter, starting by passing a new specification type and specification values to fdesign.halfband.

```
 hs = fdesign.halfband('n,ast',80,70);
hd =design(hs,'equiripple'); % Opens FVTool automatically.
```

In this example, pass the specifications for the filter, and then design a least-squares FIR filter from the object, using firls as the design method.

```
 hs = fdesign.halfband('n,tw', 42, .04);
% designmethods(hs)
hd=design(hs,'firls');
```

Create two equiripple halfband filters with and without a nonnegative zero phase response:

```
f=fdesign.halfband('N,TW',12,0.2);
% Equiripple halfband filter with nonnegative zero phase response
Hd1=design(f,'equiripple','ZeroPhase',true);
% Equiripple halfband filter with zero phase false
% 'zerophase',false is the default
Hd2=design(f,'equiripple','ZeroPhase',false);
%Obtain real-valued amplitudes (not magnitudes)
[Hr_zerophase,W]=zerophase(Hd1);
```

```
[Hr,W]=zerophase(Hd2);
% Plot and compare response
plot(W,Hr_zerophase,'k','linewidth',2);
xlabel('Radians/sample'); ylabel('Amplitude');
hold on;
plot(W,Hr,'r');
axis tight; grid on;
legend('with ''ZeroPhase'', true','with ''ZeroPhase'' false');
```

Note that the amplitude of the zero phase response (black line) is nonnegative for all frequencies.
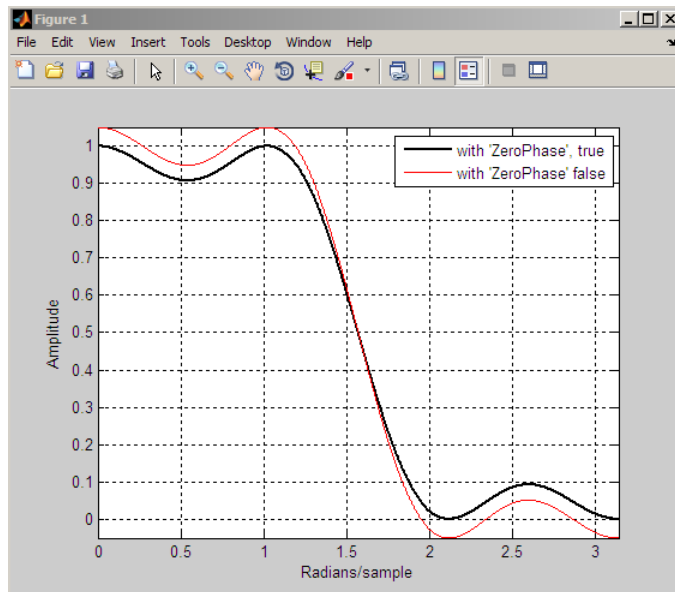
The 'ZeroPhase' option is valid only for equiripple halfband designs with the 'N,TW' specification. You cannot specify 'MinPhase' and 'ZeroPhase' to be simultaneously 'true'.

See Also   fdesign | fdesign.decimator | design | fdesign.interpolator | fdesign.nyquist | setspecs | zerophase

fdesign.highpass

Purpose Highpass filter specification object

Syntax
```
D = fdesign.highpass
D = fdesign.highpass(SPEC)
D = fdesign.highpass(SPEC,specvalue1,specvalue2,...)
D = fdesign.highpass(specvalue1,specvalue2,specvalue3,
specvalue4)
D = fdesign.highpass(...,Fs)
D = fdesign.highpass(...,MAGUNITS)
```

Description D = fdesign.highpass constructs a highpass filter specification object D, applying default values for the specification string, 'Fst,Fp,Ast,Ap'.

D = fdesign.highpass(SPEC) constructs object D and sets the Specification property to SPEC. Entries in the SPEC string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for SPEC are shown below. The strings are not case sensitive.

Note Specifications strings marked with an asterisk require the DSP System Toolbox software.

- 'Fst,Fp,Ast,Ap' (default spec)
- 'N,F3db'
- 'N,F3db,Ap' *
- 'N,F3db,Ast' *
- 'N,F3db,Ast,Ap' *
- 'N,F3db,Fp *
- 'N,Fc'
- 'N,Fc,Ast,Ap'
- 'N,Fp,Ap'

- `'N,Fp,Ast,Ap'`

- `'N,Fst,Ast'`

- `'N,Fst,Ast,Ap'`

- `'N,Fst,F3db'` *

- `'N,Fst,Fp'`

- `'N,Fst,Fp,Ap'` *

- `'N,Fst,Fp,Ast'` *

- `'Nb,Na,Fst,Fp'` *

The string entries are defined as follows:

- `Ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.

- `Ast` — attenuation in the stop band in decibels (the default units). Also called Astop.

- `F3db` — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.

- `Fc` — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.

- `Fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.

- `Fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.

- `N` — filter order.

- `Na` and `Nb` are the order of the denominator and numerator.

Graphically, the filter specifications look similar to those shown in the following figure.

Regions between specification values like `Fst` and `Fp` are transition regions where the filter response is not explicitly defined.

The filter design methods that apply to a highpass filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design method applies to an object and its specification string.

Use `designopts` to determine which design options are valid for a given design method. For detailed information on design options for a given design method, `METHOD`, enter `help(D,METHOD)` at the MATLAB command line.

`D = fdesign.highpass(SPEC,specvalue1,specvalue2,...)` constructs an object d and sets its specification values at construction time.

`D = fdesign.highpass(specvalue1,specvalue2,specvalue3, specvalue4)` constructs an object `D` with the default `Specification` property and the values you enter for `specvalue1,specvalue2,...`.

`D = fdesign.highpass(...,Fs)` provides the sampling frequency for the filter specification object. `Fs` is in Hz and must be specified as a scalar trailing the other numerical values provided. If you specify a sampling frequency, all other frequency specifications are in Hz.

`D = fdesign.highpass(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units

- `'dB'` — specify the magnitude in dB (decibels)

- `'squared'` — specify the magnitude in power units

When you omit the MAGUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   Higpass filter a discrete-time signal consisting of two sine waves.

Create a highpass filter specification object. Specify the passband frequency to be 0.25π radians/sample and the stopband frequency to be 0.15π radians/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
d = fdesign.highpass('Fst,Fp,Ast,Ap',0.15,0.25,60,1);
```

Query the valid design methods for your filter specification object, d.

```
designmethods(d)
```

Create an FIR equiripple filter and view the filter magnitude response with fvtool.

```
Hd = design(d,'equiripple');
fvtool(Hd);
```

Create a signal consisting of the sum of two discrete-time sinusoids with frequencies of $\pi/8$ and $\pi/4$ radians/sample and amplitudes of 1 and 0.25 respectively. Filter the discrete-time signal with the FIR equiripple filter object, Hd

```
n = 0:159;
x = cos((pi/8)*n)+0.25*sin((pi/4)*n);
y = filter(Hd,x);
Domega = (2*pi)/160;
freq = 0:(2*pi)/160:pi;
```

```
xdft = fft(x);
ydft = fft(y);
plot(freq,abs(xdft(1:length(x)/2+1)));
hold on;
plot(freq,abs(ydft(1:length(y)/2+1)),'r','linewidth',2);
legend('Original Signal','Lowpass Signal', ...
'Location','NorthEast');
ylabel('Magnitude'); xlabel('Radians/Sample');
```

Create a filter of order 10 with a 6-dB frequency of 9.6 kHz and a sampling frequency of 48 kHz.

```
d=fdesign.highpass('N,Fc',10,9600,48000);
designmethods(d)
% only valid design method is FIR window method
Hd = design(d);
% Display filter magnitude response
fvtool(Hd);
```

If you have the DSP System Toolbox software, you can specify the shape of the stopband and the rate at which the stopband decays.

Create two FIR equiripple filters with different linear stopband slopes. Specify the passband frequency to be 0.3π radians/sample and the stopband frequency to be 0.35π radians/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB. Design one filter with a 20 dB/rad/sample stopband slope and another filter with 40 dB/rad/sample.

```
D = fdesign.highpass('Fst,Fp,Ast,Ap',0.3,0.35,60,1);
Hd1 = design(D,'equiripple','StopBandShape','linear','StopBandDecay',20);
Hd2 = design(D,'equiripple','StopBandShape','linear','StopBandDecay',40);
hfvt = fvtool([Hd1 Hd2]);
legend(hfvt,'20 dB/rad/sample','40 dB/rad/sample');
```

See Also   design | designmethods | fdesign

fdesign.hilbert

Purpose   Hilbert filter specification object

Syntax
```
d = fdesign.hilbert
d = fdesign.hilbert(specvalue1,specvalue2)
d = fdesign.hilbert(spec)
d = fdesign.hilbert(spec,specvalue1,specvalue2)
d = fdesign.hilbert(...,Fs)
d = fdesign.hilbert(...,MAGUNITS)
```

Description   d = fdesign.hilbert constructs a default Hilbert filter designer d with N, the filter order, set to 30 and TW, the transition width set to 0.1π radians/sample.

d = fdesign.hilbert(specvalue1,specvalue2) constructs a Hilbert filter designer d assuming the default specification string 'N,TW'. You input specvalue1 and specvalue2 for N and TW.

d = fdesign.hilbert(spec) initializes the filter designer Specification property to spec. You provide one of the following strings as input to replace spec. The specification strings are not case sensitive.

Note Specifications strings marked with an asterisk require the DSP System Toolbox software.

- 'N,TW' default spec string.

- 'TW,Ap' *

The string entries are defined as follows:

- Ap — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.

- N — filter order.

- TW — width of the transition region between the pass and stop bands.

By default, `fdesign.hilbert` assumes that all frequency specifications are provided in normalized frequency units. Also, decibels is the default for all magnitude specifications.

Different specification strings may have different design methods available. Use `designmethods(d)` to get a list of the design methods available for a given specification string.

`d = fdesign.hilbert(`*spec*`,specvalue1,specvalue2)` initializes the filter designer specifications in spec with specvalue1, specvalue2, and so on. To get a description of the specifications specvalue1 and specvalue2, enter

`get(d,'`**description**`')`

at the Command prompt.

`d = fdesign.hilbert(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.hilbert(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units
- `'dB'` — specify the magnitude in dB (decibels)
- `'squared'` — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   Design a Hilbert transformer of order 30 with a transition width of 0.2π radians/sample. Plot the zero phase response from [-π,π) radians/sample and the impulse response.

```
d = fdesign.hilbert('N,TW',30,0.2);
```

```
% Show available design methods
designmethods(d)
% Use least square minimization to obtain linear-phase FIR filter
Hd = design(d,'equiripple');
% Display zero phase response from [-pi,pi)
fvtool(Hd,'magnitudedisplay','zero-phase',...
'frequencyrange','[-pi, pi)')
```
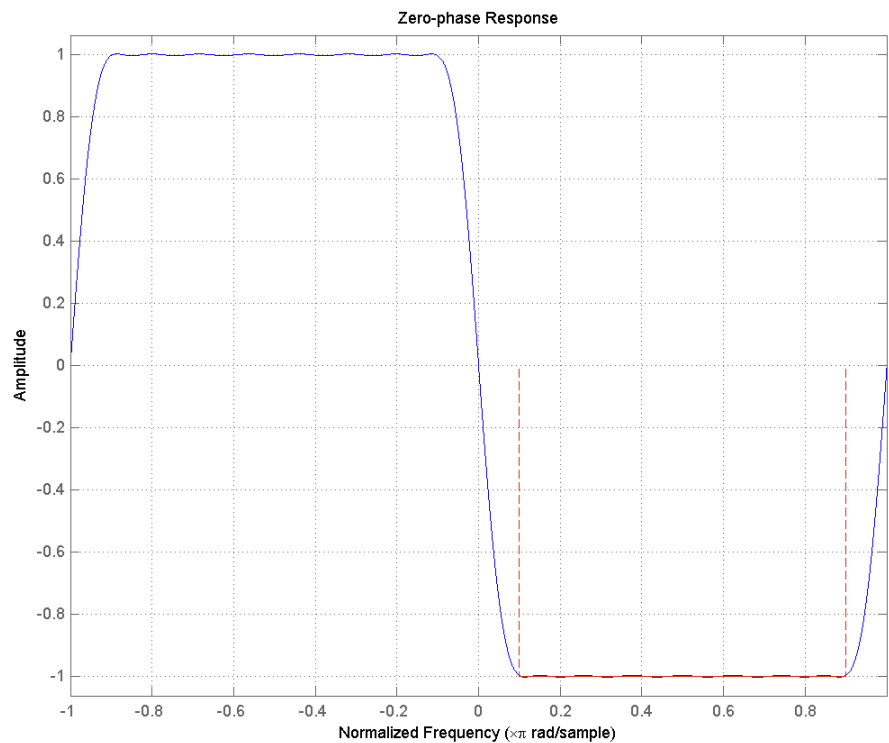

The impulse response of this even order filter is antisymmetric (type III).

```
fvtool(Hd,'analysis','impulse')
```


Impulse Response
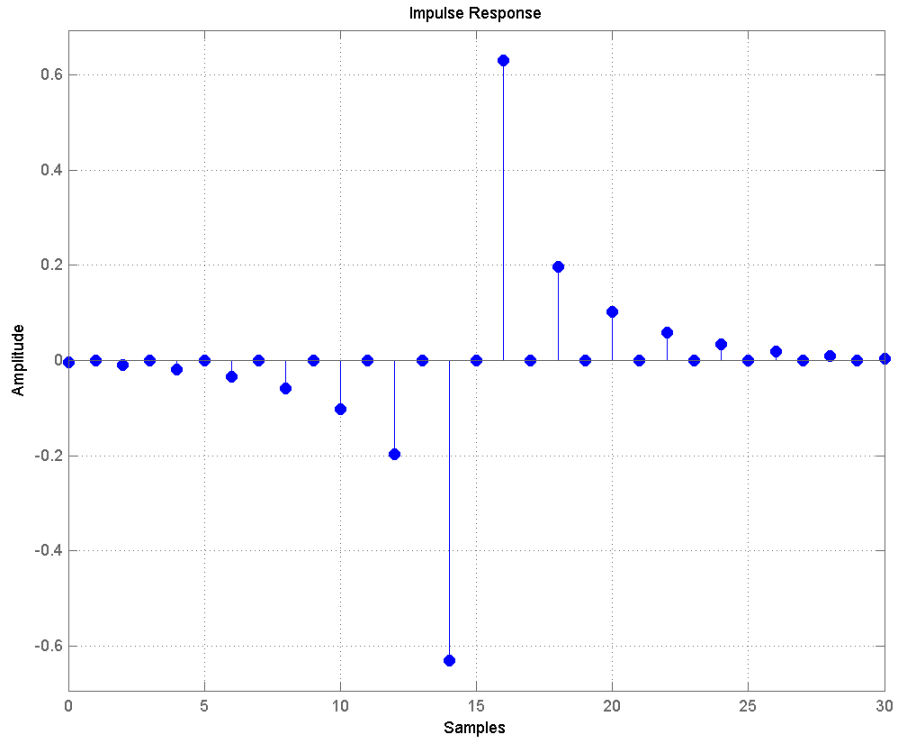
Apply the filter to a discrete-time sinusoid with a frequency of π/2 radians/sample.

```
n = 0:99;
x = cos(pi/2*n);
y = filter(Hd,x);
% Correct for the filter delay
Delay = floor(length(Hd.Numerator)/2);
y = y(Delay+1:end);
```

Plot a the filter input and output and validate the approximate π/2 phase shift obtained with the Hilbert transformer.

```
stem(x(1:end-Delay),'markerfacecolor',[0 0 1]);
hold on;
stem(y,'Color',[1 0 0],'markerfacecolor',[1 0 0]);
axis([10 20 -1.5 1.5]); grid on;
xlabel('Samples'); ylabel('Amplitude');
legend('Input','Output','Location','NorthWest')
```
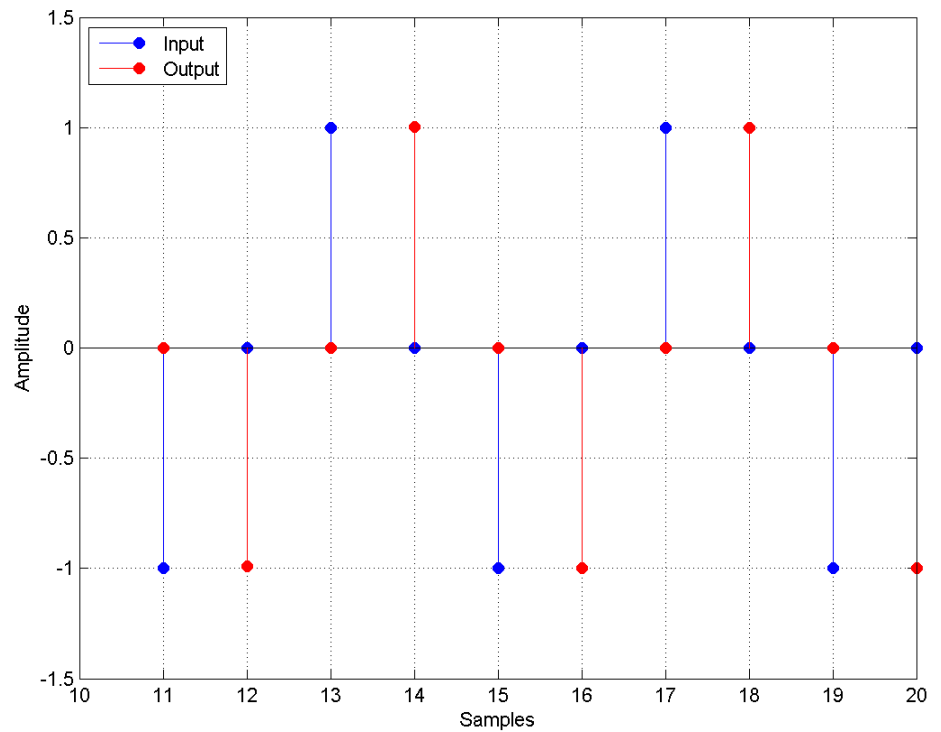
Because the frequency of the discrete-time sinusoid is π/2 radians/sample, a one sample shift corresponds to a phase shift of π/2.

Form the analytic signal and demonstrate that the frequency content of the analytic signal is zero for negative frequencies and approximately twice the spectrum of the input for positive frequencies.

```
x1 = x(1:end-Delay);
% Form the analytic signal
xa = x1+1j*y;
freq = -pi:(2*pi)/length(x1):pi-(2*pi)/length(x);
plot(freq,abs(fftshift(fft(x1))));
hold on;
plot(freq,abs(fftshift(fft(xa))),'r'); grid on;
xlabel('Radians/Sample'); ylabel('Magnitude');
legend('Input DFT','Analytic Signal DFT','Location','NorthWest');
```
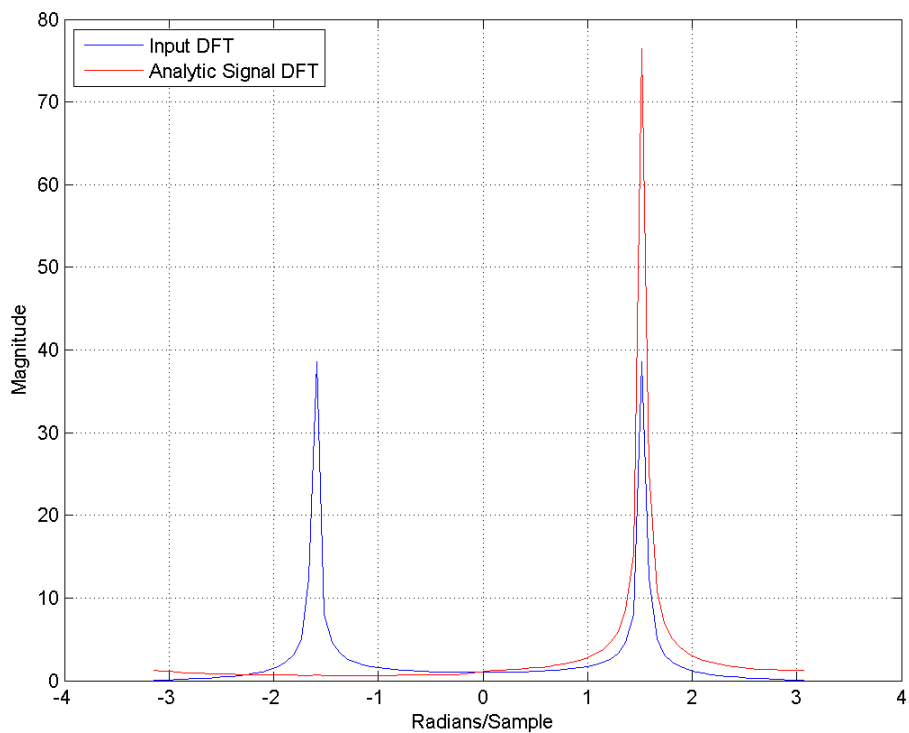
Design a minimum-order Hilbert transformer that has a sampling frequency of 1 kHz. Specify the passband ripple to be 1 dB.

```
d = fdesign.hilbert('TW,Ap',1,0.1,1e3);
hd = design(d,'equiripple');
fvtool(hd,'magnitudedisplay','zero-phase', ...
'frequencyrange','[-Fs/2, Fs/2)');
```
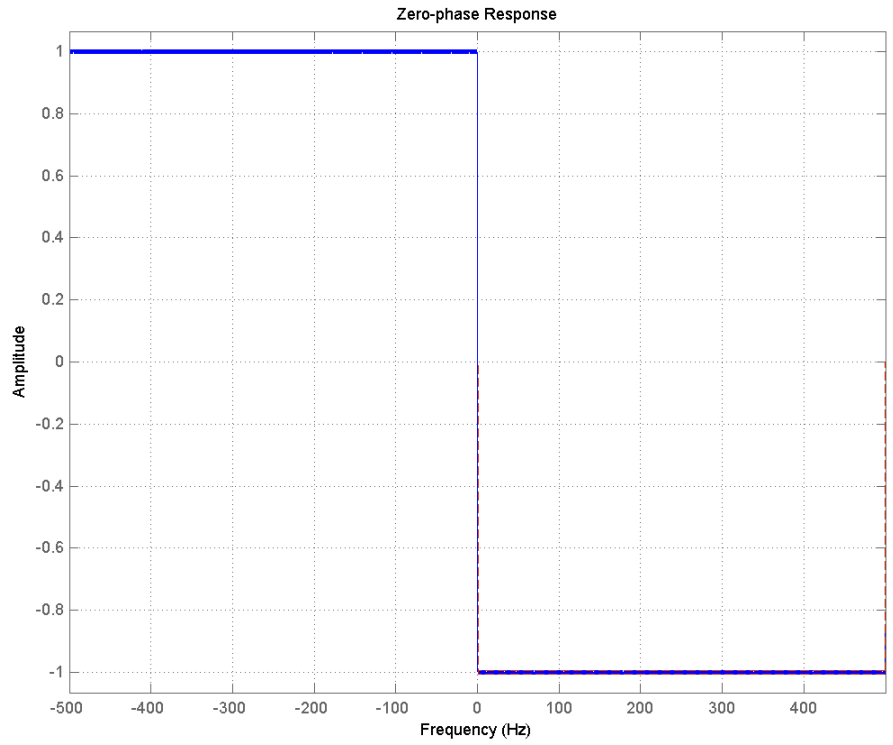
See Also   design | fdesign | setspecs

fdesign.interpolator

Purpose	Interpolator filter specification
Syntax	D = fdesign.interpolator(L)
	D = fdesign.interpolator(L,RESPONSE)
	D = fdesign.interpolator(L,PULSESHAPINGRESPONSE,SPS)
	D = fdesign.interpolator(L,CICRESPONSE,D)
	D = fdesign.interpolator(L,RESPONSE,spec)
	D = fdesign.interpolator(...,spec,specvalue1,specvalue2,...)
	D = fdesign.interpolator(...,Fs)
	d = fdesign.interpolator(...,MAGUNITS)

Description   D = fdesign.interpolator(L) constructs an interpolator filter
specification object D with the InterpolationFactor property equal to
the positive integer L and the Response property set to 'Nyquist'. The
default values for the transition width and stopband attenuation in the
Nyquist design are 0.1π radians/sample and 80 dB. If L is unspecified,
L defaults to 2.

D = fdesign.interpolator(L,RESPONSE) constructs a interpolator
specification object with the interpolation factor L and the 'Response'
property set to one of the supported types.

D = fdesign.interpolator(L,PULSESHAPINGRESPONSE,SPS) constructs
a pulse-shaping interpolator specification object with the interpolation
factor L and 'Response' property equal to one of the supported
pulse-shaping filters: 'Gaussian', 'Raised Cosine', or 'Square Root
Raised Cosine'. SPS is the samples per symbol. The samples per
symbol, SPS, must precede any specification string.

D = fdesign.interpolator(L,CICRESPONSE,D) constructs a CIC or
CIC compensator interpolator specification object with the interpolation
factor, L, and 'Response' property equal to 'CIC' or 'CICCOMP'. D
is the differential delay. The differential delay, D, must precede any
specification string.

D = fdesign.interpolator(L,RESPONSE,spec) constructs object D and
sets its Specification property to spec. Entries in the spec string
represent various filter response features, such as the filter order, that

govern the filter design. Valid entries for spec depend on the design type of the specifications object.

When you add the spec input argument, you must also add the RESPONSE input argument.

Because you are designing multirate filters, the specification strings available are not the same as the specifications for designing single-rate filters with design methods such as fdesign.lowpass. The strings are not case sensitive.

The interpolation factor L is not in the specification strings. The different filter responses support different specifications. The following table lists the supported response types and specification strings.

Design String	Valid Specification Strings
'Arbitrary Magnitude'	See fdesign.arbmag for a description of the specification string entries. • 'N,F,A' (default string) • 'N,B,F,A'
'Arbitrary Magnitude and Phase'	See fdesign.arbmagnphase for a description of the specification string entries. • 'N,F,H' (default string) • 'N,B,F,H'
'Bandpass'	See fdesign.bandpass for a description of the specification string entries. • 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2' (default string) • 'N,Fc1,Fc2' • 'N,Fst1,Fp1,Fp2,Fst2'
'Bandstop'	See fdesign.bandstop for a description of the specification string entries. • 'N,Fc1,Fc2' • 'N,Fp1,Fst1,Fst2,Fp2' • 'Fp1,Fst1,Fst2,Fp2,Ap1,Ast,Ap2' (default string)

Design String	Valid Specification Strings
`'CIC'`	`'Fp,Ast'` — Only valid specification. `Fp` is the passband frequency and `Ast` is the stopband attenuation in decibels.
	To specify a CIC interpolator, include the differential delay after `'CIC'` and before the filter specification string: `'Fp,Ast'`. For example: `d = fdesign.interpolator(2,'cic',4,'Fp,Ast',0.4,40);`
`'CIC Compensator'`	See `fdesign.ciccomp` for a description of the specification string entries.
	• `'Fp,Fst,Ap,Ast'` (default string)
	• `'N,Fc,Ap,Ast'`
	• `'N,Fp,Ap,Ast'`
	• `'N,Fp,Fst'`
	• `'N,Fst,Ap,Ast'`
	To specify a CIC compensator interpolator, include the differential delay after `'CICCOMP'` and before the filter specification string.. For example: `d = fdesign.interpolator(2,'ciccomp',4);`
`'Differentiator'`	`'N'` — filter order
`'Gaussian'`	`'Nsym,BT` — `Nsym` is the filter order in symbols and `BT` is the bandwidth-symbol time product.
	The specification string must be preceded by an integer-valued `SamplesPerSymbol`.

Design String	Valid Specification Strings
'Halfband	See fdesign.halfband for a description of the specification string entries. • 'TW,Ast' (default string) • 'N,TW' • `N' • 'N,Ast' If you use the quasi-linear IIR design method, iirlinphase, with a halfband specification, the interpolation factor must be 2.
'Highpass'	See fdesign.highpass for a description of the specification string entries. • 'Fst,Fp,Ast,Ap' (default string) • 'N,F3db' • 'N,Fc' • 'N,Fc,Ast,Ap' • 'N,Fp,Ast,Ap' • 'N,Fst,Ast,Ap' • 'N,Fst,Fp' • 'N,Fst,Ast,Ap' • 'N,Fst,Fp,Ast'
'Hilbert'	See fdesign.hilbert for a description of the specification string entries. • 'N,TW' (default string) • 'TW,Ap'

Design String	Valid Specification Strings
`'Inverse-sinc Lowpass'`	See `fdesign.isinclp` for a description of the specification string entries. • `'Fp,Fst,Ap,Ast'` (default string)) • `'N,Fc,Ap,Ast'` • `'N,Fp,Fst'` • `'N,Fst,Ap,Ast'`
`'Inverse-sinc Highpass'`	See `fdesign.isinchp` for a description of the specification string entries. • `'Fst,Fp,Ast,Ap'` (default string)) • `'N,Fc,Ast,Ap'` • `'N,Fst,Fp'` • `'N,Fst,Ast,Ap'`
`'Lowpass'`	See `fdesign.lowpass` for a description of the specification string entries. • `'Fp,Fst,Ap,Ast'` (default string) • `'N,F3dB'` • `'N,Fc'` • `'N,Fc,Ap,Ast'` • `'N,Fp,Ap,Ast'` • `'N,Fp,Fst'` • `'N,Fp,Fst,Ap'` • `'N,Fp,Fst,Ast'` • `'N,Fst,Ap,Ast'`

Design String	Valid Specification Strings
'Nyquist'	See fdesign.nyquist for a description of the specification string entries. For all Nyquist specifications, you must specify the *L*th band. This typically corresponds to the interpolation factor so that the nonzero samples of the upsampler output are preserved. • 'TW,Ast' (default string) • 'N' • 'N,Ast' • 'N,Ast'
'Raised Cosine'	See fdesign.pulseshaping for a description of the specification string entries. • 'Ast,Beta' (default string) • 'Nsym,Beta' • 'N,Beta' The specification string must be preceded by an integer-valued SamplesPerSymbol.
'Square Root Raised Cosine'	See fdesign.pulseshaping for a description of the specification string entries. • 'Ast,Beta' (default string) • 'Nsym,Beta' • 'N,Beta' The specification string must be preceded by an integer-valued SamplesPerSymbol.

D = fdesign.interpolator(...,spec,specvalue1,specvalue2,...) constructs an object D and sets its specifications at construction time.

D = fdesign.interpolator(...,Fs) adds the argument Fs, specified in Hz, to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

d = fdesign.interpolator(...,MAGUNITS) specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

• 'linear' — specify the magnitude in linear units.

• 'dB' — specify the magnitude in dB (decibels).

• 'squared' — specify the magnitude in power units.

When you omit the MAGUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   These examples show how to construct interpolating filter specification objects. First, create a default specifications object without using input arguments except for the interpolation factor l.

```
l = 2;
d = fdesign.interpolator(2);
```

Now create an object by passing a specification string 'fst1,fp1,fp2,fst2,ast1,ap,ast2' and a design — the resulting object uses default values for all of the filter specifications. You must provide the design input argument when you include a specification.

```
d=fdesign.interpolator(8,'bandpass','fst1,fp1,fp2,fst2,ast1,ap,ast2');
```

Create another interpolating filter object, passing the specification values to the object rather than accepting the default values for, in this case, fp,fst,ap,ast.

```
d=fdesign.interpolator(3,'lowpass',.45,0.55,.1,60);
```

Now pass the filter specifications that correspond to the specifications — n,fc,ap,ast.

```
d=fdesign.interpolator(3,'ciccomp',1,2,'n,fc,ap,ast',...
20,0.45,.05,50);
```

With the specifications object in your workspace, design an interpolator using the equiripple design method.

```
hm = design(d,'equiripple');
```

Pass a new specification type for the filter, specifying the filter order.

```
d = fdesign.interpolator(5,'CIC',1,'fp,ast',0.55,55);
```

With the specifications object in your workspace, design an interpolator using the multisection design method.

```
hm = design(d,'multisection');
```

In this example, you specify a sampling frequency as the right most input argument. Here, it is set to 1000 Hz.

```
d=fdesign.interpolator(8,'bandpass','fst1,fp1,fp2,fst2,ast1,ap,ast2',
    0.25,0.35,.55,.65,50,.05,1e3);
```

In this, the last example, use the linear option for the filter specification object and specify the stopband ripple attenuation in linear form.

```
d = fdesign.interpolator(4,'lowpass','n,fst,ap,ast',15,0.55,.05,...
    1e3,'linear'); % 1e3 = 60dB.
```

Now design a CIC interpolator for a signal sampled at 19200 Hz. Specify the differential delay of 2 and set the attenuation of information beyond 50 Hz to be at least 80 dB.

The filter object sampling frequency is (l x fs) where fs is the sampling frequency of the input signal.

```
dd  = 2;    % Differential delay.
fp  = 50;   % Passband of interest.
ast = 80;   % Minimum attenuation of alias components in passband.
fs  = 600;  % Sampling frequency for input signal.
l   = 32;   % Interpolation factor.
d   = fdesign.interpolator(l,'cic',dd,'fp,ast',fp,ast,l*fs);
hm = design(d);  %Use the default design method.
```

This next example results in a minimum-order CIC compensator that interpolates by 4 and compensates for the droop in the passband for the CIC filter hm from the previous example.

```
nsecs = hm.numberofsections;
d = fdesign.interpolator(4,'ciccomp',dd,nsecs,...
50,100,0.1,80,fs);
hmc = design(d,'equiripple');
hmc.arithmetic = 'fixed';
```

hmc is designed to compensate for hm. To see the effect of the compensating CIC filter, use FVTool to analyze both filters individually and include the compound filter response by cascading hm and hmc.

```
hfvt = fvtool(hmc,hm,cascade(hmc,hm),'fs',[fs,l*fs,l*fs],...
'showreference','off');
legend(hfvt,'CIC Compensator','CIC Interpolator',...
'Overall Response');
```

FVTool returns with this plot.
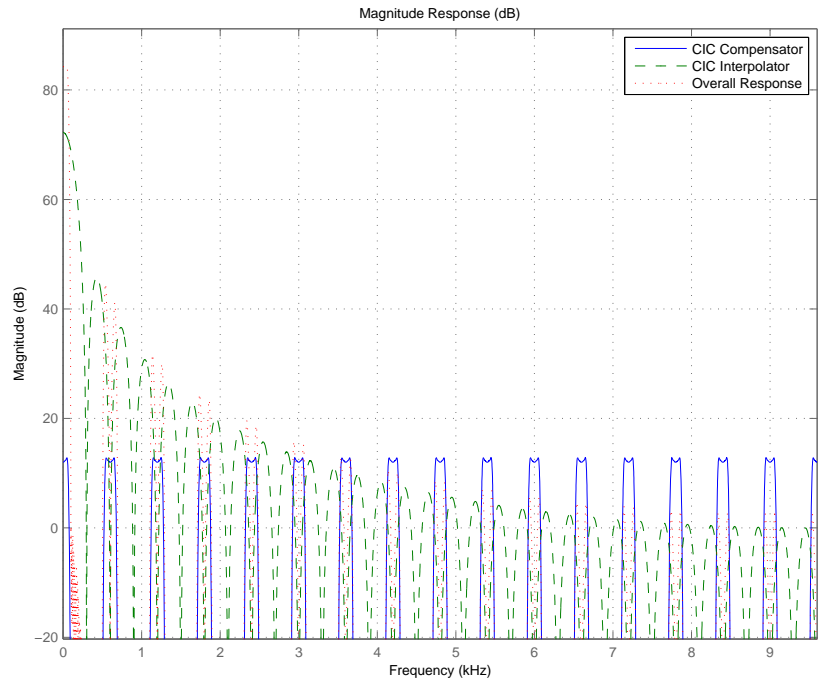
For the third example, use `fdesign.interpolator` to design a minimum-order Nyquist interpolator that uses a Kaiser window. For comparison, design a multistage interpolator as well and compare the responses.

```
l   = 15;   % Set the interpolation factor and the Nyquist band.
tw  = 0.05; % Specify the normalized transition width.
ast = 40;   % Set the minimum stopband attenuation in dB.
d   = fdesign.interpolator(l,'nyquist',l,tw,ast);
hm  = design(d,'kaiserwin');
hm2 = design(d,'multistage'); % Design the multistage interpolator.
hfvt = fvtool(hm,hm2);
legend(hfvt,'Kaiser Window','Multistage')
```

FVTool shows both responses.
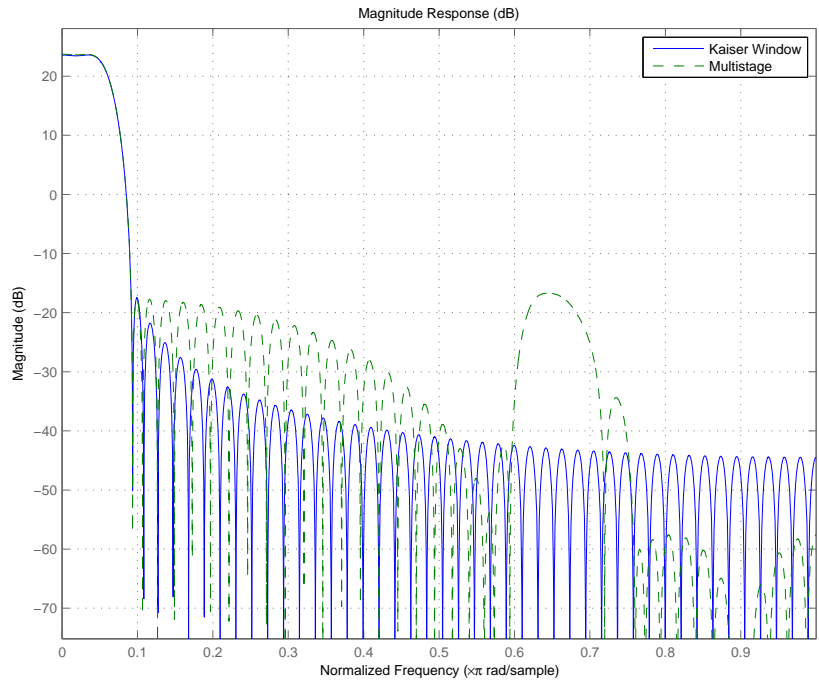
Design a lowpass interpolator for an interpolation factor of 8. Compare the single-stage equiripple design to a multistage design with the same interpolation factor.

```
l = 8; % Interpolation factor.
d = fdesign.interpolator(l,'lowpass');
hm(1) = design(d,'equiripple');
% Use halfband filters whenever possible.
hm(2) = design(d,'multistage','usehalfbands',true);
hfvt = fvtool(hm);
legend(hfvt,'Single-Stage Equiripple','Multistage')
```
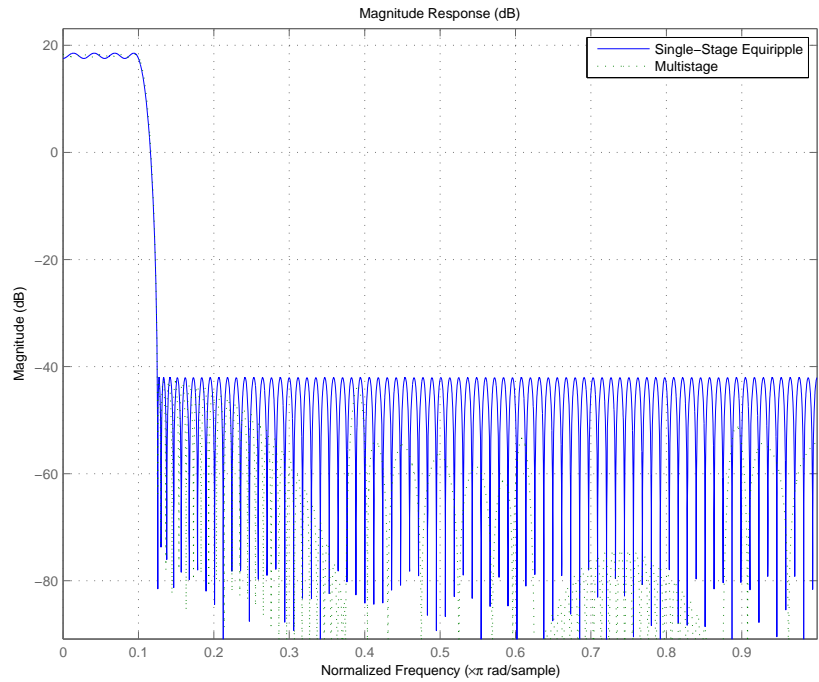
See Also   fdesign | fdesign.arbmag | fdesign.arbmagnphase |
fdesign.interpolator | fdesign.rsrc | setspecs

fdesign.isinchp

Purpose   Inverse sinc highpass filter specification

Syntax
```
D = fdesign.isinchp
D = fdesign.isinchp(SPEC)
D = fdesign.isinchp(SPEC,specvalue1,specvalue2,...)
D = fdesign.isinchp(specvalue1,specvalue2,specvalue3,
   specvalue4)
D = fdesign.isinchp(...,Fs)
D = fdesign.isinchp(...,MAGUNITS)
```

Description   `D = fdesign.isinchp` constructs an inverse sinc highpass filter specification object `D`, applying default values for the default specification string `'Fst,Fp,Ast,Ap'`.

`D = fdesign.isinchp(SPEC)` constructs object `D` and sets the `Specification` property to SPEC. Entries in the SPEC string represent various filter response features, such as the filter order, that govern the filter design. Valid entries for SPEC are shown below. The strings are not case sensitive.

- `'Fst,Fp,Ast,Ap'` (default spec)

- `'N,Fc,Ast,Ap'`

- `'N,Fst,Fp'`

- `'N,Fp,Ast,Ap'`

- `'N,Fst,Ast,Ap'`

The string entries are defined as follows:

- `Ast` — attenuation in the stopband in decibels (the default units). Also called Astop.

- `Ap` — amount of ripple allowed in the passband in decibels (the default units). Also called Apass.

- `Fp` — frequency at the start of the passband. Specified in normalized frequency units. Also called Fpass.

- Fst — frequency at the end of the stopband. Specified in normalized frequency units. Also called Fstop.

- N — filter order.

The filter design methods that apply to an inverse sinc highpass filter specification object change depending on the value of the Specification property. Use designmethods to determine which design method applies to a specific Specification.

Use designopts to see the available design options for a specific design method. Enter help(D,METHOD) at the MATLAB command line to obtain detailed information on the design options for a given design method, METHOD.

D = fdesign.isinchp(SPEC,specvalue1,specvalue2,...) constructs an object D and sets the specifications at construction time.

D = fdesign.isinchp(specvalue1,specvalue2,specvalue3,specvalue4) constructs an object D assuming the default Specification property string 'Fst,Fp,Ast,Ap', using the values you provide in specvalue1,specvalue2, specvalue3, and specvalue4.

D = fdesign.isinchp(...,Fs) adds the argument Fs, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

D = fdesign.isinchp(...,MAGUNITS) specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

- 'linear' — specify the magnitude in linear units

- 'dB' — specify the magnitude in dB (decibels)

- 'squared' — specify the magnitude in power units

When you omit the MAGUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

fdesign.isinchp

The design method of fdesign.isinchp implements a filter with a passband magnitude response equal to:

$$H(\omega) = sinc(C(1-\omega))^{-P}$$

You can control the values of the sinc frequency factor, *C*, and the sinc power, *P*, using the 'SincFrequencyFactor' and 'SincPower' options in the design method. 'SincFrequencyFactor' and 'SincPower' default to 0.5 and 1 respectively.

Examples   Design a minimum order inverse sinc highpass filter and shape the stopband to have a slope of 20 dB/radian/sample.

```
d = fdesign.isinchp('Fst,Fp,Ast,Ap',.4,.5,40,0.01);
Hd = design(d);
% Shape the stopband to have a linear slope of 20 dB/rad/sample
Hd1 = design(d,'StopbandShape','linear','StopbandDecay',20);
fvtool(Hd,Hd1);
```
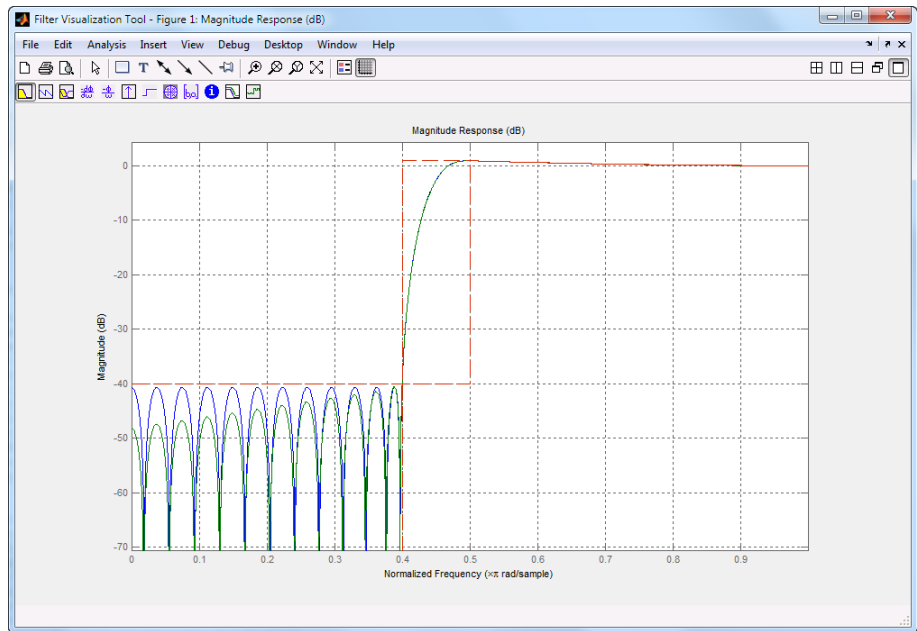
Design a 50th order inverse sinc highpass filter. Set the sinc frequency factor to 0.25, and the sinc power to 16 to achieve a magnitude response in the passband of the form $H(\omega) = \text{sinc}(0.25*(1-\omega))^{(-16)}$.

```
d = fdesign.isinchp('N,Fst,Fp',50,.4,.5);
Hd = design(d,'SincFrequencyFactor',0.25,'SincPower',16);
fvtool(Hd);
```
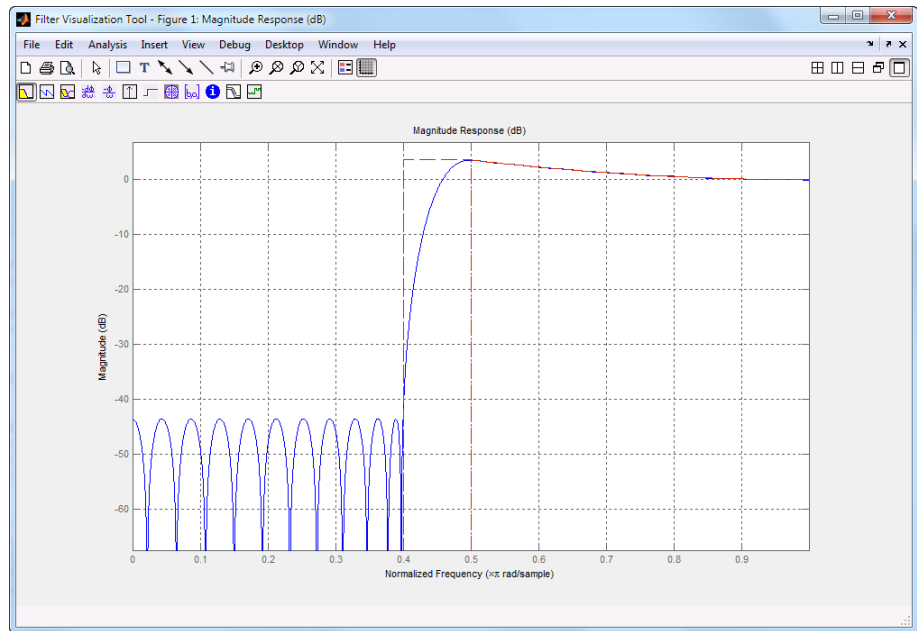
See Also   design | designmethods | fdesign | fdesign.ciccomp |
fdesign.highpass | fdesign.isinclp | fdesign.nyquist

Purpose   Inverse sinc lowpass filter specification

Syntax
```
d = fdesign.isinclp
d = fdesign.isinclp(spec)
d = fdesign.isinclp(spec,specvalue1,specvalue2,...)
d = fdesign.isinclp(specvalue1,specvalue2,specvalue3,
    specvalue4)
d = fdesign.isinclp(...,Fs)
d = fdesign.isinclp(...,MAGUNITS)
```

Description   d = fdesign.isinclp constructs an inverse sinc lowpass filter
specification object d, applying default values for the default
specification string, 'Fp,Fst,Ap,Ast'.

d = fdesign.isinclp(*spec*) constructs object d and sets its
'Specification' to *spec*. Entries in the spec string represent various
filter response features, such as the filter order, that govern the filter
design. Valid entries for *spec* are shown below. The strings are not
case sensitive.

- 'Fp,Fst,Ap,Ast' (default spec)

- 'N,Fc,Ap,Ast'

- 'N,Fp,Ap,Ast'

- 'N,Fp,Fst'

- 'N,Fst,Ap,Ast'

The string entries are defined as follows:

- Ast — attenuation in the stopband in decibels (the default units).
  Also called Astop.

- Ap — amount of ripple allowed in the passband in decibels (the
  default units). Also called Apass.

- Fp — frequency at the start of the passband. Specified in normalized
  frequency units. Also called Fpass.

- `Fst` — frequency at the end of the stopband. Specified in normalized frequency units. Also called Fstop.

- `N` — filter order.

The filter design methods that apply to an inverse sinc lowpass filter specification object change depending on the `Specification` string. Use `designmethods` to determine which design method applies to an object and its specification string.

`d = fdesign.isinclp(spec,specvalue1,specvalue2,...)` constructs an object `d` and sets its specifications at construction time.

`d = fdesign.isinclp(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `d` assuming the default `Specification` property string `'Fp,Fst,Ap,Ast'`, using the values you provide in `specvalue1,specvalue2, specvalue3,` and `specvalue4`.

`d = fdesign.isinclp(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`d = fdesign.isinclp(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units

- `'dB'` — specify the magnitude in dB (decibels)

- `'squared'` — specify the magnitude in power units

When you omit the `MAGUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

The design method of `fdesign.isinclp` implements a filter with a passband magnitude response equal to:

$$H(\omega) = sinc(C\omega)^{-P}$$

You can control the values of the sinc frequency factor, *C*, and the sinc power, *P*, using the `'SincFrequencyFactor'` and `'SincPower'` options in the design method. `'SincFrequencyFactor'` and `'SincPower'` default to 0.5 and 1 respectively.

Examples   Pass the specifications for the default specification — `'Fp,Fst,Ap,Ast'` — as input arguments to the specifications object.

```
d = fdesign.isinclp(.4,.5,.01,40);
hd = design(d,'equiripple');
fvtool(hd);
```

Design a 50th order inverse sinc lowpass filter. Set the sinc frequency factor to 0.25 and the sinc power to 16 to achieve a magnitude response in the passband of the form `H(w) = sinc(0.25*w)^(-16)` .

```
d = fdesign.isinclp('N,Fp,Fst',50,.4,.5);
Hd = design(d,'SincFrequencyFactor',0.25,'SincPower',16);
fvtool(Hd, 'MagnitudeDisplay', 'Magnitude');
```

fdesign.isinclp
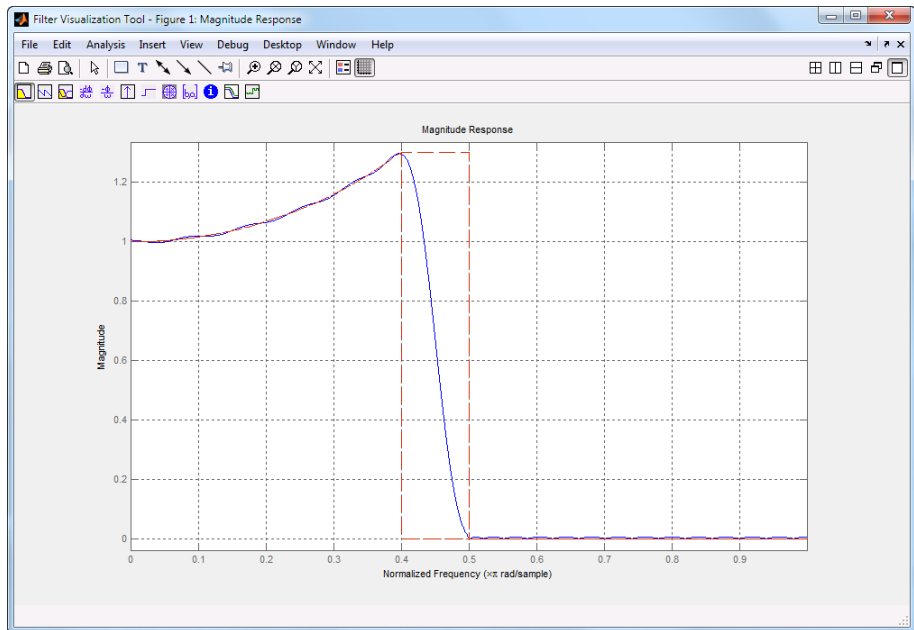
See Also   fdesign | fdesign.bandpass | fdesign.bandstop |
fdesign.halfband | fdesign.highpass | fdesign.lowpass |
fdesign.nyquist

Purpose   Lowpass filter specification

Syntax   D = fdesign.lowpass
                 D = fdesign.lowpass(SPEC)
                 D = fdesign.lowpass(SPEC,specvalue1,specvalue2,...)
                 D = fdesign.lowpass(specvalue1,specvalue2,specvalue3,
                     specvalue4)
                 D = fdesign.lowpass(...,Fs)
                 D = fdesign.lowpass(...,MAGUNITS)

Description   D = fdesign.lowpass constructs a lowpass filter specification
                 object D, applying default values for the default specification string
                 'Fp,Fst,Ap,Ast'.

                 D = fdesign.lowpass(SPEC) constructs object D and sets the
                 Specification property to the string in SPEC. Entries in the SPEC
                 string represent various filter response features, such as the filter
                 order, that govern the filter design. Valid entries for SPEC are shown
                 below. The strings are not case sensitive.

                 Note Specifications strings marked with an asterisk require the DSP
                 System Toolbox software.

                 • 'Fp,Fst,Ap,Ast' (default spec)

                 • 'N,F3db'

                 • 'N,F3db,Ap' *

                 • 'N,F3db,Ap,Ast' *

                 • 'N,F3db,Ast' *

                 • 'N,F3db,Fst' *

                 • 'N,Fc'

                 • "N,Fc,Ap,Ast'

- `'N,Fp,Ap'`
- `'N,Fp,Ap,Ast'`
- `'N,Fp,Fst,Ap'` *
- `'N,Fp,F3db'` *
- `'N,Fp,Fst'`
- `'N,Fp,Fst,Ast'` *
- `'N,Fst,Ap,Ast'` *
- `'N,Fst,Ast'`
- `'Nb,Na,Fp,Fst'` *

The string entries are defined as follows:

- `Ap` — amount of ripple allowed in the pass band in decibels (the default units). Also called Apass.

- `Ast` — attenuation in the stop band in decibels (the default units). Also called Astop.

- `F3db` — cutoff frequency for the point 3 dB point below the passband value. Specified in normalized frequency units.

- `Fc` — cutoff frequency for the point 6 dB point below the passband value. Specified in normalized frequency units.

- `Fp` — frequency at the start of the pass band. Specified in normalized frequency units. Also called Fpass.

- `Fst` — frequency at the end of the stop band. Specified in normalized frequency units. Also called Fstop.

- `N` — filter order.

- `Na` and `Nb` are the order of the denominator and numerator.

Graphically, the filter specifications look similar to those shown in the following figure.

Regions between specification values like `Fp` and `Fst` are transition regions where the filter response is not explicitly defined.

`D = fdesign.lowpass(SPEC,specvalue1,specvalue2,...)` constructs an object `D` and sets the specification values at construction time using `specvalue1`, `specvalue2`, and so on for all of the specification variables in `SPEC`.

`D = fdesign.lowpass(specvalue1,specvalue2,specvalue3,specvalue4)` constructs an object `D` with values for the default `Specification` property string `'Fp,Fst,Ap,Ast'` using the specifications you provide as input arguments `specvalue1,specvalue2,specvalue3,specvalue4`.

`D = fdesign.lowpass(...,Fs)` adds the argument `Fs`, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

`D = fdesign.lowpass(...,MAGUNITS)` specifies the units for any magnitude specification you provide in the input arguments. `MAGUNITS` can be one of

- `'linear'` — specify the magnitude in linear units

- `'dB'` — specify the magnitude in dB (decibels)

- `'squared'` — specify the magnitude in power units

fdesign.lowpass

When you omit the `MAGNUNITS` argument, `fdesign` assumes that all magnitudes are in decibels. Note that `fdesign` stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   Lowpass filter a discrete-time signal consisting of two sine waves.

Create a lowpass filter specification object. Specify the passband frequency to be 0.15π radians/sample and the stopband frequency to be 0.25π radians/sample. Specify 1 dB of allowable passband ripple and a stopband attenuation of 60 dB.

```
d=fdesign.lowpass('Fp,Fst,Ap,Ast',0.15,0.25,1,60);
```

Query the valid design methods for your filter specification object, `d`.

```
designmethods(d)
```

Create an FIR equiripple filter and view the filter magnitude response with `fvtool`.

```
Hd = design(d,'equiripple');
fvtool(Hd);
```

Create a signal consisting of the sum of two discrete-time sinusoids with frequencies of $\pi/8$ and $\pi/4$ radians/sample and amplitudes of 1 and 0.25 respectively. Filter the discrete-time signal with the FIR equiripple filter object, `Hd`.

```
n = 0:159;
x = 0.25*cos((pi/8)*n)+sin((pi/4)*n);
y = filter(Hd,x);
Domega = (2*pi)/160;
freq = 0:(2*pi)/160:pi;
xdft = fft(x);
ydft = fft(y);
plot(freq,abs(xdft(1:length(x)/2+1)));
hold on;
plot(freq,abs(ydft(1:length(y)/2+1)),'r','linewidth',2);
```

```
legend('Original Signal','Highpass Signal', ...
'Location','NorthEast');
ylabel('Magnitude'); xlabel('Radians/Sample');
```

Create a filter of order 10 with a 6-dB frequency of 9.6 kHz and a sampling frequency of 48 kHz.

```
d=fdesign.lowpass('N,Fc',10,9600,48000);
designmethods(d)
% only valid design method is FIR window method
Hd = design(d);
% Display filter magnitude response
fvtool(Hd);
```

Zoom in on the magnitude response to verify that the -6 dB point is at 9.6 kHz.

If you have the DSP System Toolbox software, you can specify the shape of the stopband and the rate at which the stopband decays. The following example requires the DSP System Toolbox.

Create an FIR equiripple filter with a passband frequency of 0.2π radians/sample, a stopband frequency of 0.25π radians/sample, a passband ripple of 1 dB, and a stopband attenuation of 60 dB. Design the filter with a 20 dB/rad/sample linear stopband.

```
D = fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.25,1,60);
Hd = design(D,'equiripple','StopBandShape','linear','StopBandDecay',20
fvtool(Hd);
```

See Also   design | designmethods | fdesign

fdesign.notch

Purpose   Notch filter specification

Syntax
```
d = fdesign.notch(specstring, value1, value2, ...)
d = fdesign.notch(n,f0,q)
d = fdesign.notch(...,Fs)
d = fdesign.notch(...,MAGUNITS)
```

Description   `d = fdesign.notch(specstring, value1, value2, ...)` constructs a notch filter specification object `d`, with a specification string set to `specstring` and values provided for all members of the `specstring`. The possible specification strings, which are not case sensitive, are listed as follows:

- 'N,F0,Q' (default)

- 'N,F0,Q,Ap'

- 'N,F0,Q,Ast'

- 'N,F0,Q,Ap,Ast'

- 'N,F0,BW'

- 'N,F0,BW,Ap'

- 'N,F0,BW,Ast'

- 'N,F0,BW,Ap,Ast'

where the variables are defined as follows:

- N - Filter Order (must be even)

- F0 - Center Frequency

- Q - Quality Factor

- BW - 3-dB Bandwidth

- Ap - Passband Ripple (decibels)

- Ast - Stopband Attenuation (decibels)

Different specification strings, resulting in different specification objects, may have different design methods available. Use the function

designmethods to get a list of design methods available for a given specification. For example:

```
d  = fdesign.notch('N,F0,Q,Ap',6,0.5,10,1);
% designmethods(d) returns
% Design Methods for class fdesign.notch (N,F0,Q,Ap):
% cheby1
```

d = fdesign.notch(n,f0,q) constructs a notch filter specification object using the default specstring ('N,F0,Q') and setting the corresponding values to n, f0, and q.

By default, all frequency specifications are assumed to be in normalized frequency units. All magnitude specifications are assumed to be in decibels.

d = fdesign.notch(...,Fs) constructs a notch filter specification object while providing the sampling frequency of the signal to be filtered. Fs must be specified as a scalar trailing the other values provided. If you specify an Fs, it is assumed to be in Hz, as are all the other frequency values provided.

d = fdesign.notch(...,MAGUNITS) constructs a notch filter specification while providing the units for any magnitude specification given. MAGUNITS can be one of the following: 'linear', 'dB', or 'squared'. If this argument is omitted, 'dB' is assumed. The magnitude specifications are always converted and stored in decibels regardless of how they were specified. If Fs is provided, MAGUNITS must follow Fs in the input argument list.

Examples   Design a notching filter with a passband ripple of 1 dB.

```
d  = fdesign.notch('N,F0,Q,Ap',6,0.5,10,1);
Hd = design(d);
fvtool(Hd)
```
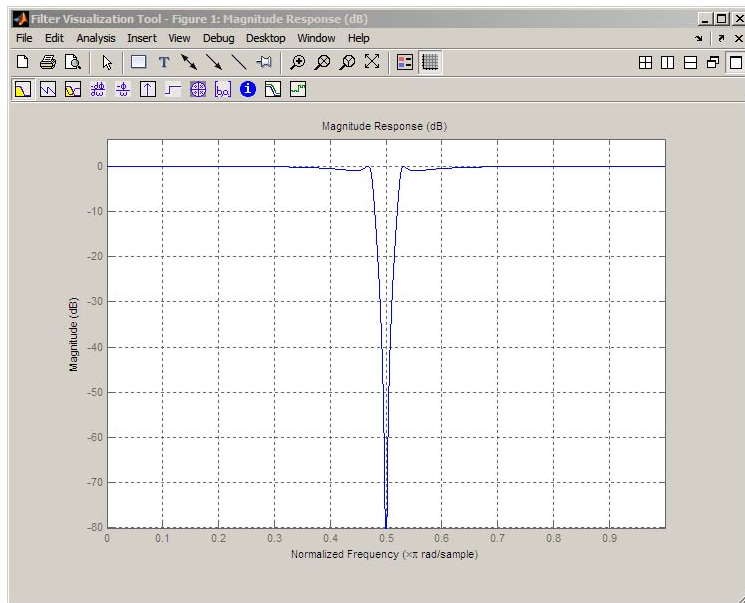
See Also   `fdesign | fdesign.peak`

Purpose   Nyquist filter specification

Syntax   d = fdesign.nyquist
                     d = fdesign.nyquist(l,*spec*)
                     d = fdesign.nyquist(l,*spec*,specvalue1,specvalue2,...)
                     d = fdesign.nyquist(l,specvalue1,specvalue2)
                     d = fdesign.nyquist(...,fs)
                     d = fdesign.nyquist(...,magunits)

Description   d = fdesign.nyquist constructs a Nyquist or L-band filter specification
                     object d, applying default values for the properties tw and ast. By
                     default, the filter object designs a minimum-order half-band (L=2)
                     Nyquist filter.

                     Using fdesign.nyquist with a design method generates a dfilt object.

                     d = fdesign.nyquist(l,*spec*) constructs object d and sets its
                     Specification property to *spec*. Use l to specify the desired value for
                     L. L = 2 designs a half-band FIR filter, L = 3 a third-band FIR filter,
                     and so on. When you use a Nyquist filter as an interpolator, l or L is the
                     interpolation factor. The first input argument must be l when you are
                     not using the default syntax d = fdesign.nyquist.

                     Entries in the spec string represent various filter response features,
                     such as the filter order, that govern the filter design. Valid entries for
                     spec are shown below. The strings are not case sensitive.

                     - tw,ast (default spec)

                     - n,tw

                     - n

                     - n,ast

                     The string entries are defined as follows:

                     - ast — attenuation in the stop band in decibels (the default units).

                     - n — filter order.

- tw — width of the transition region between the pass and stop bands. Specified in normalized frequency units.

The filter design methods that apply to a Nyquist filter specification object change depending on the Specification string. Use designmethods to determine which design method applies to an object and its specification string. Different filter design methods also have options that you can specify. Use designopts with the design method string to see the available options. For example:

```
f=fdesign.nyquist(4,'N,TW');
designmethods(f)
```

d = fdesign.nyquist(l,*spec*,specvalue1,specvalue2,...) constructs an object d and sets its specification to *spec*, and the specification values to specvalue1, specvalue2, and so on at construction time.

d = fdesign.nyquist(l,specvalue1,specvalue2) constructs an object d with the values you provide in l, specvalue1,specvalue2 as the values for l, tw and ast.

d = fdesign.nyquist(...,fs) adds the argument fs, specified in Hz to define the sampling frequency to use. In this case, all frequencies in the specifications are in Hz as well.

d = fdesign.nyquist(...,magunits) specifies the units for any magnitude specification you provide in the input arguments. magunits can be one of

- linear — specify the magnitude in linear units
- dB — specify the magnitude in dB (decibels)
- squared — specify the magnitude in power units

When you omit the magunits argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Limitations of the Nyquist fdesign Object

Using Nyquist filter specification objects with the `equiripple` design method imposes a few limitations on the resulting filter, caused by the `equiripple` design algorithm.

- When you request a minimum-order design from `equiripple` with your Nyquist object, the design algorithm might not converge and can fail with a filter convergence error.

- When you specify the order of your desired filter, and use the `equiripple` design method, the design might not converge.

- Generally, the following specifications, alone or in combination with one another, can cause filter convergence problems with Nyquist objects and the `equiripple` design method.

  - very high order

  - small transition width

  - very large stopband attenuation

Note that halfband filters (filters where band = 2) do not exhibit convergence problems.

When convergence issues arise, either in the cases mentioned or in others, you might be able to design your filter with the `kaiserwin` method.

In addition, if you use Nyquist objects to design decimators or interpolators (where the interpolation or decimation factor is not a prime number), using multistage filter designs might be your best approach.

Examples   These examples show how to construct a Nyquist filter specification object. First, create a default specifications object without using input arguments.

```
d=fdesign.nyquist
```

Now create an object by passing a specification type string 'n,ast' — the resulting object uses default values for n and ast.

```
d=fdesign.nyquist(2,'n,ast')
```

Create another Nyquist filter object, passing the specification values to the object rather than accepting the default values for n and ast.

```
d=fdesign.nyquist(3,'n,ast',42,80)
```

Finally, pass the filter specifications that correspond to the default Specification — tw,ast. When you pass only the values, fdesign.nyquist assumes the default Specification string.

```
d = fdesign.nyquist(4,.01,80)
```

Now design a Nyquist filter using the kaiserwin design method.

```
hd = design(d,'kaiserwin')
```

Create two equiripple Nyquist 4th–band filters with and without a nonnegative zero phase response:

```
f=fdesign.nyquist(4,'N,TW',12,0.2);
% Equiripple Nyquist 4th-band filter with
% nonnegative zero phase response
Hd1=design(f,'equiripple','zerophase',true);
% Equiripple Nyquist 4th-band filter with 'ZeroPhase' set to false
% 'zerophase',false is the default
Hd2=design(f,'equiripple','zerophase',false);
%Obtain real-valued amplitudes (not magnitudes)
[Hr_zerophase,W]=zerophase(Hd1);
[Hr,W]=zerophase(Hd2);
% Plot and compare response
plot(W,Hr_zerophase,'k','linewidth',2);
xlabel('Radians/sample'); ylabel('Amplitude');
hold on;
plot(W,Hr,'r');
axis tight; grid on;
```

```
legend('with ''ZeroPhase'', true','with ''ZeroPhase'' false');
```

Note that the amplitude of the zero phase response (black line) is nonnegative for all frequencies.
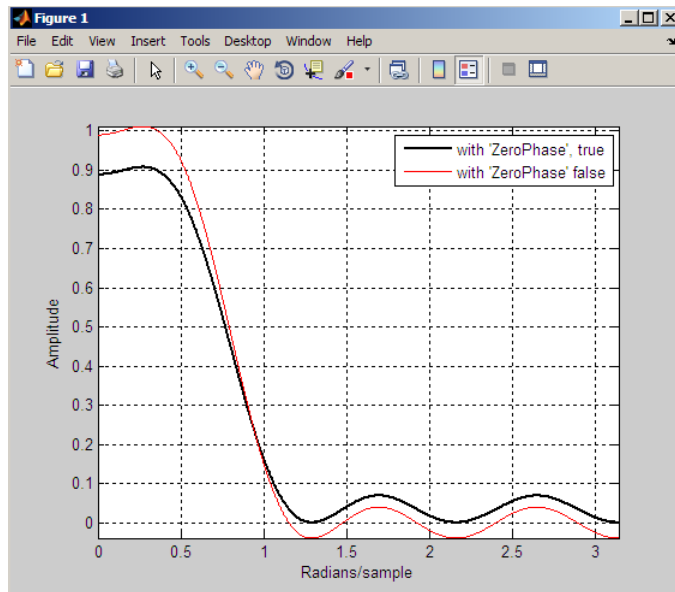
The 'ZeroPhase' option is valid only for equiripple Nyquist designs with the 'N,TW' specification. You cannot specify 'MinPhase' and 'ZeroPhase' to be simultaneously 'true'.

See Also fdesign | fdesign.interpolator | fdesign.halfband | fdesign.interpolator | fdesign.rsrc | zerophase

fdesign.octave

Purpose　　Octave filter specification

Syntax　　
```
d = fdesign.octave(l)
d = fdesign.octave(l, MASK)
d = fdesign.octave(l, MASK, spec)
d = fdesign.octave(..., Fs)
```

Description　　`d = fdesign.octave(l)` constructs an octave filter specification object `d`, with `l` bands per octave. The default value for `l` is 1.

`d = fdesign.octave(l, MASK)` constructs an octave filter specification object `d` with `l` bands per octave and `MASK` specification for the FVTool. The available values for mask are:

- `'class 0'`

- `'class 1'`

- `'class 2'`

`d = fdesign.octave(l, MASK, spec)` constructs an octave filter specification object `d` with `l` bands per octave, `MASK` specification for the FVTool, and the `spec` specification string. The specification strings available are:

- `'N, F0'`
(not case sensitive), where:

- `N` is the filter order

- `F0` is the center frequency.

The center frequency is typically specified in normalized frequency units, unless a sampling frequency in Hz is included in the specification: `d = fdesign.octave(..., Fs)`. In this case, all frequencies must be specified in Hz, with the center frequency falling between 20 Hz and 20 kHz (the audio range).

Examples　　Design an sixth order, octave-band class 0 filter with a center frequency of 1000 Hz and, a sampling frequency of 44.1 kHz.

```
d = fdesign.octave(1,'Class 0','N,F0',6,1000,44100);
Hd = design(d);
fvtool(Hd)
```

The following figure shows the magnitude response plot of the filter. The logarithmic scale for frequency is automatically set by FVTool for the octave filters.
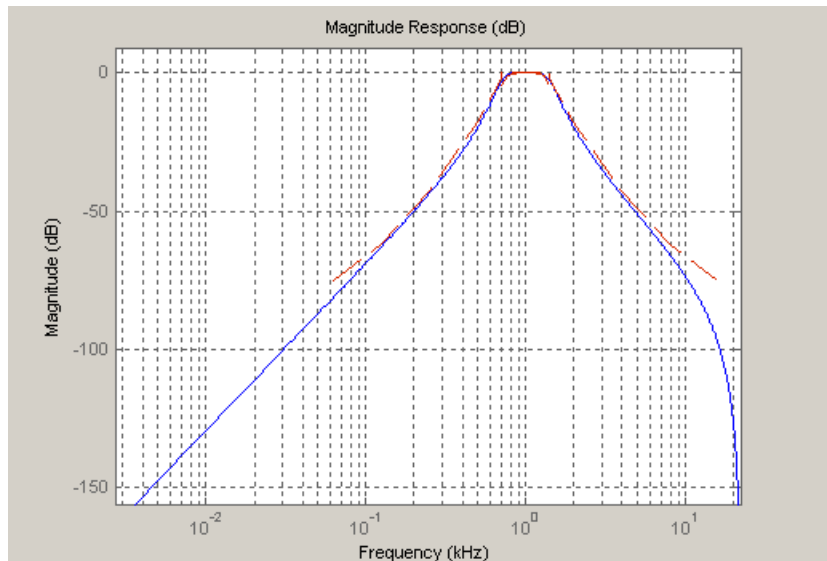
See Also   fdesign

fdesign.parameq

Purpose	Parametric equalizer filter specification
Syntax	d = fdesign.parameq(spec, specvalue1, specvalue2, ...) d = fdesign.parameq(... fs)
Description	d = fdesign.parameq(spec, specvalue1, specvalue2, ...) constructs a parametric equalizer filter design object, where spec is a non-case sensitive specification string. The choices for spec are as follows:

- 'FO, BW, BWp, Gref, GO, GBW, Gp' (minimum order default)

- 'FO, BW, BWst, Gref, GO, GBW, Gst'

- 'FO, BW, BWp, Gref, GO, GBW, Gp, Gst'

- 'N, FO, BW, Gref, GO, GBW'

- 'N, FO, BW, Gref, GO, GBW, Gp'

- 'N, FO, Fc, Qa, GO'

- 'N, FO, Fc, S, GO'

- 'N, FO ,BW, Gref, GO, GBW, Gst'

- 'N, FO, BW, Gref, GO, GBW, Gp, Gst'

- 'N, Flow, Fhigh, Gref, GO, GBW'

- 'N, Flow, Fhigh, Gref, GO, GBW, Gp'

- 'N, Flow, Fhigh, Gref, GO, GBW, Gst'

- 'N, Flow, Fhigh, Gref, GO, GBW, Gp, Gst'

where the parameters are defined as follows:

- BW — Bandwidth

- BWp — Passband Bandwidth

- BWst — Stopband Bandwidth

- Gref — Reference Gain (decibels)

- G0 — Center Frequency Gain (decibels)

- GBW — Gain at which Bandwidth (BW) is measured (decibels)

- Gp — Passband Gain (decibels)

- Gst — Stopband Gain (decibels)

- N — Filter Order

- F0 — Center Frequency

- Fc— Cutoff frequency

- Fhigh - Higher Frequency at Gain GBW

- Flow - Lower Frequency at Gain GBW

- Qa-Quality Factor

- S-Slope Parameter for Shelving Filters

Regardless of the specification string chosen, there are some conditions that apply to the specification parameters. These are as follows:

- Specifications for parametric equalizers must be given in decibels

- To boost the input signal, set `G0 > Gref`; to cut, set `Gref > G0`

- For boost: `G0 > Gp > GBW > Gst > Gref`; For cut: `G0 < Gp < GBW < Gst < Gref`

- Bandwidth must satisfy: `BWst > BW > BWp`

`d = fdesign.parameq(...  fs)` adds the input sampling frequency. Fs must be specified as a scalar trailing the other numerical values provided, and is assumed to be in Hz.

Examples   Design a Chebyshev Type II parametric equalizer filter that cuts by 12 dB:

```
d = fdesign.parameq('N,Flow,Fhigh,Gref,G0,GBW,Gst',...
     4,.3,.5,0,-12,-10,-1);
Hd = design(d,'cheby2');
fvtool(Hd)
```
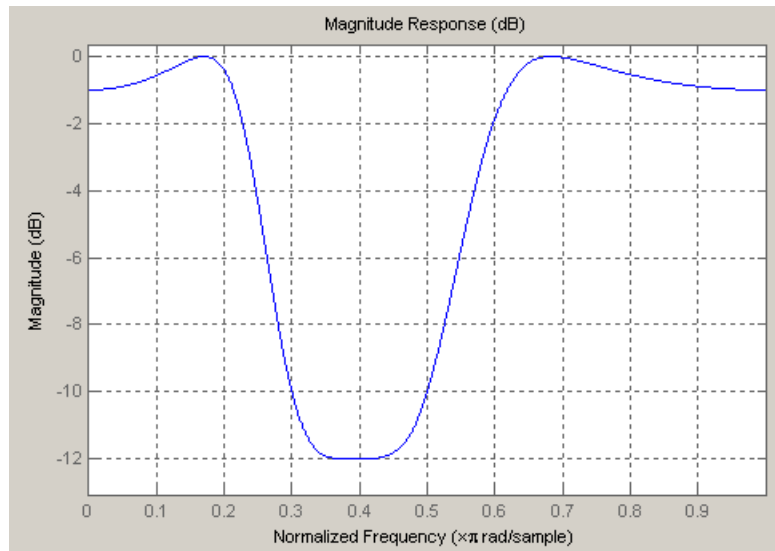
Design a 4th order audio lowpass (F0 = 0) shelving filter with cutoff frequency of Fc = 0.25, quality factor Qa =10, and boost gain of G0 = 8 dB:

```
d = fdesign.parameq('N,F0,Fc,Qa,G0',4,0,0.25,10,8);
Hd = design(d);
fvtool(Hd)
```
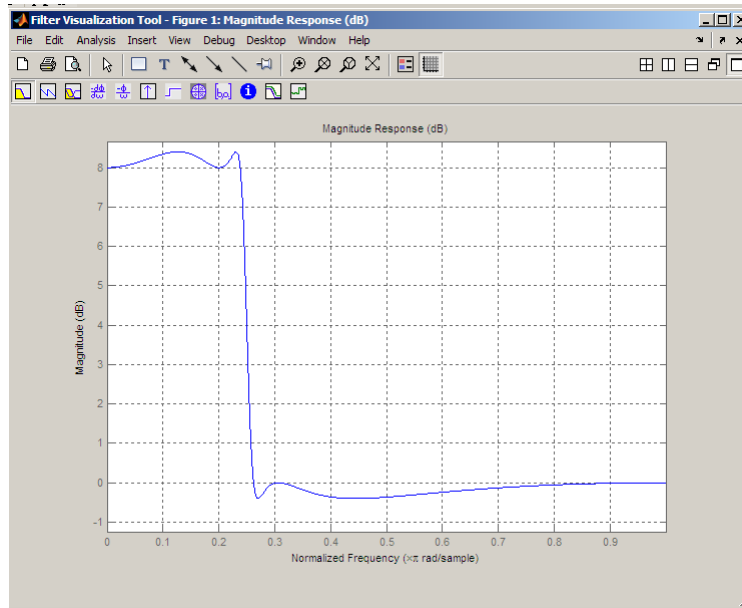
Design 4th-order highpass shelving filters with S=1.5 and S=3:

```
N=4;
FO = 1;
Fc = .4; % Cutoff Frequency
GO = 10;
S = 1.5;
S2=3;
f = fdesign.parameq('N,FO,Fc,S,GO',N,FO,Fc,S,GO);
h1 = design(f);
f.S=3;
h2=design(f);
hfvt=fvtool([h1 h2]);
set(hfvt,'Filters',[h1 h2]);
legend(hfvt,'S=1.5','S=3');
```
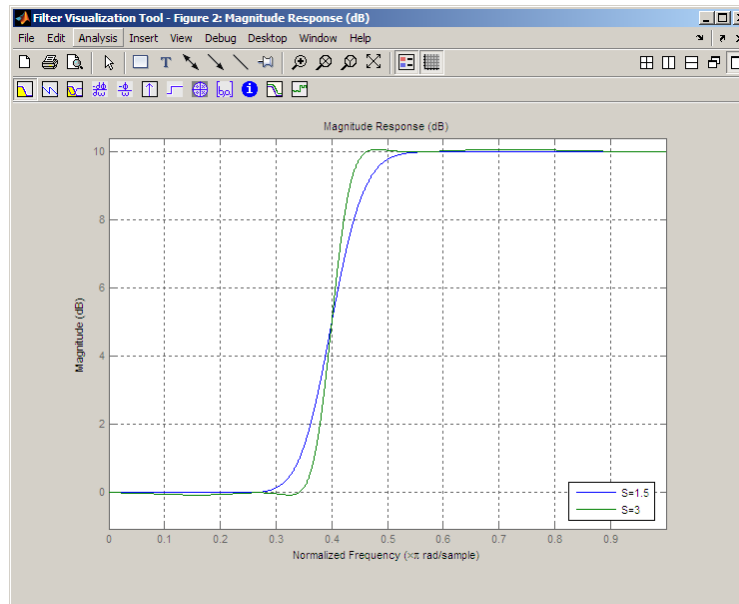
See Also   fdesign

How To   • Parametric Equalizer Design

fdesign.peak

Purpose   Peak filter specification

Syntax
```
d = fdesign.peak(specstring, value1, value2, ...)
d = fdesign.peak(n,f0,q)
d = fdesign.peak(...,Fs)
d = fdesign.peak(...,MAGUNITS)
```

Description   `d = fdesign.peak(specstring, value1, value2, ...)` constructs a peaking filter specification object `d`, with a specification string set to `specstring` and values provided for all members of the `specstring`. The possible specification strings, which are not case sensitive, are listed as follows:

- 'N,F0,Q' (default)
- 'N,F0,Q,Ap'
- 'N,F0,Q,Ast'
- 'N,F0,Q,Ap,Ast'
- 'N,F0,BW'
- 'N,F0,BW,Ap'
- 'N,F0,BW,Ast'
- 'N,F0,BW,Ap,Ast'

where the variables are defined as follows:

- N - Filter Order (must be even)
- F0 - Center Frequency
- Q - Quality Factor
- BW - 3-dB Bandwidth
- Ap - Passband Ripple (decibels)
- Ast - Stopband Attenuation (decibels)

Different specification strings, resulting in different specification objects, may have different design methods available. Use the function

7-619

designmethods to get a list of design methods available for a given specification. For example:

```
>> d  = fdesign.peak('N,F0,Q,Ap',6,0.5,10,1);
>> designmethods(d)


Design Methods for class fdesign.peak (N,F0,Q,Ap):


cheby1
```

d = fdesign.peak(n,f0,q) constructs a peaking filter specification object using the default specstring ('N,F0,Q') and setting the corresponding values to n, f0, and q.

By default, all frequency specifications are assumed to be in normalized frequency units. All magnitude specifications are assumed to be in decibels.

d = fdesign.peak(...,Fs) constructs a peak filter specification object while providing the sampling frequency of the signal to be filtered. Fs must be specified as a scalar trailing the other values provided. If you specify an Fs, it is assumed to be in Hz, as all the other frequency values provided.

d = fdesign.peak(...,MAGUNITS) constructs a notch filter specification while providing the units for any magnitude specification given. MAGUNITS can be one of the following: 'linear', 'dB', or 'squared'. If this argument is omitted, 'dB' is assumed. The magnitude specifications are always converted and stored in decibels regardless of how they were specified. If Fs is provided, MAGUNITS must follow Fs in the input argument list.

Examples   Design a Chebyshev Type II peaking filter with a stopband attenuation of 80 dB:

```
d = fdesign.peak('N,F0,BW,Ast',8,.65,.02,80);
Hd = design(d,'cheby2');
```

```
fvtool(Hd)
```

This design produces a filter with the magnitude response shown in the following figure.
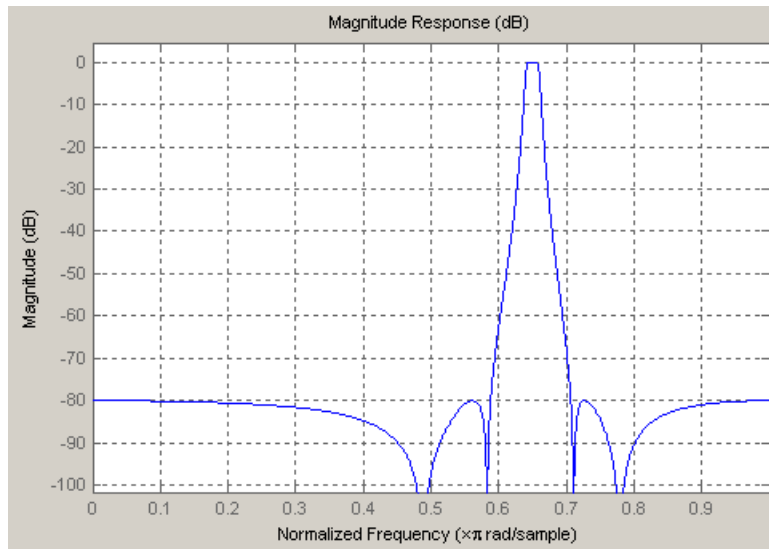
See Also   fdesign | fdesign.notch | fdesign.parameq

fdesign.polysrc

Purpose
Construct polynomial sample-rate converter (POLYSRC) filter designer

Syntax
```
d = fdesign.polysrc(l,m)
d = fdesign.polysrc(l,m,'Fractional Delay','Np',Np)
d = fdesign.polysrc(...,Fs)
```

Description
d = fdesign.polysrc(l,m) constructs a polynomial sample-rate converter filter designer D with an interpolation factor L and a decimation factor M. L defaults to 3. M defaults to 2. L and M can be arbitrary positive numbers.

d = fdesign.polysrc(l,m,'Fractional Delay','Np',Np) initializes the filter designer specification with Np and sets the polynomial order to the value Np. If omitted Np defaults to 3.

d = fdesign.polysrc(...,Fs) specifies the sampling frequency (in Hz).

Examples
This example shows how to design sample-rate converter that uses a 3rd order Lagrange interpolation filter to convert from 44.1kHz to 48kHz:

```
[L,M] = rat(48/44.1);
f = fdesign.polysrc(L,M,'Fractional Delay','Np',3);
Hm = design(f,'lagrange');
% Original sampling frequency
Fs = 44.1e3;
% 9408 samples, 0.213 seconds long
n = 0:9407;
% Original signal, sinusoid at 1kHz
x  = sin(2*pi*1e3/Fs*n);
% 10241 samples, still 0.213 seconds
y = filter(Hm,x);
% Plot original sampled at 44.1kHz
stem(n(1:45)/Fs,x(1:45))
hold on
% Plot fractionally interpolated signal (48kHz) in red
stem((n(3:51)-2)/(Fs*L/M),y(3:51),'r','filled')
xlabel('Time (sec)');ylabel('Signal value')
legend('44.1 kHz sample rate','48 kHz sample rate')
```

This code generates the following figure.
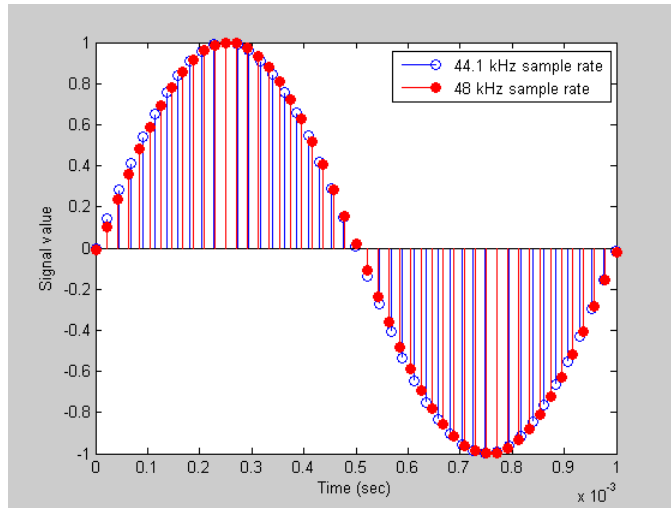
For more information about Farrow SRCs, see the "Efficient Sample Rate Conversion between Arbitrary Factors" example, efficientsrcdemo.

See Also   fdesign

fdesign.pulseshaping

Purpose   Pulse-shaping filter specification object

Syntax
```
D = fdesign.pulseshaping
D = fdesign.pulseshaping(sps)
D = fdesign.pulseshaping(sps,shape)
d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...)
d = fdesign.pulseshaping(...,fs)
d = fdesign.pulseshaping(...,magunits)
```

Description   D = fdesign.pulseshaping constructs a specification object D, which can be used to design a minimum-order raised cosine filter object with a default stop band attenuation of 60dB and a rolloff factor of 0.25.

D = fdesign.pulseshaping(sps) constructs a minimum-order raised cosine filter specification object d with a positive integer-valued oversampling factor, SamplesPerSymbol .

D = fdesign.pulseshaping(sps,shape) constructs d where shape specifies the PulseShape property. Valid entries for shape are:

- 'Raised Cosine'

- 'Square Root Raised Cosine'

- 'Gaussian'

d = fdesign.pulseshaping(sps,shape,spec,value1,value2,...) constructs d where spec defines the Specification properties. The string entries for spec specify various properties of the filter, including the order and frequency response. Valid entries for spec depend upon the shape property. For 'Raised Cosine' and 'Square Root Raised Cosine' filters, the valid entries for spec are:

- 'Ast,Beta' (minimum order; default)

- 'Nsym,Beta'

- 'N,Beta'

The string entries are defined as follows:

- `Ast` —stopband attenuation (in dB). The default stopband attenuation for a raised cosine filter is 60 dB. The default stopband attenuation for a square root raised cosine filter is 30 dB. If `Ast` is specified, the minimum-order filter is returned.

- `Beta` —rolloff factor expressed as a real-valued scalar ranging from 0 to 1. Smaller rolloff factors result in steeper transitions between the passband and stopband of the filter.

- `Nsym` —filter order in symbols. The length of the impulse response is given by `Nsym*SamplesPerSymbol+1`. The product `Nsym*SamplesPerSymbol` must be even.

- `N` —filter order (must be even). The length of the impulse response is `N+1`.

If the `shape` property is specified as `'Gaussian'`, the valid entries for `spec` are:

- `'Nsym,BT'` (default)

The string entries are defined as follows:

- `Nsym`—filter order in symbols. `Nsym` defaults to 6. The length of the filter impulse response is `Nsym*SamplesPerSymbol+1`. The product `Nsym*SamplesPerSymbol` must be even.

- `BT` —the 3–dB bandwidth-symbol time product. `BT` is a positive real-valued scalar, which defaults to 0.3. Larger values of `BT` produce a narrower pulse width in time with poorer concentration of energy in the frequency domain.

`d = fdesign.pulseshaping(...,fs)` specifies the sampling frequency of the signal to be filtered. `fs` must be specified as a scalar trailing the other numerical values provided. For this case, `fs` is assumed to be in Hz and is used for analysis and visualization.

`d = fdesign.pulseshaping(...,magunits)` specifies the units for any magnitude specification you provide in the input arguments. Valid entries for `magunits` are:

- `linear` — specify the magnitude in linear units

fdesign.pulseshaping

- dB — specify the magnitude in dB (decibels)

- squared — specify the magnitude in power units

When you omit the magunits argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

After creating the specification object d, you can use the design function to create a filter object such as h in the following example:

```
d = fdesign.pulseshaping(8,'Raised Cosine','Nsym,Beta',6,0.25);
h = design(d);
```

Normally, the Specification property of the specification object also determines which design methods you can use when you create the filter object. Currently, regardless of the Specification property, the design function uses the window design method with all fdesign.pulseshaping specification objects. The window method creates an FIR filter with a windowed impulse response.

Examples

Pulse-shaping can be used to change the waveform of transmitted pulses so the signal bandwidth matches that of the communication channel. This helps to reduce distortion and intersymbol interference (ISI).

This example shows how to design a minimum-order raised cosine filter that provides a stop band attenuation of 60 dB, rolloff factor of 0.50, and 8 samples per symbol.

```
h  = fdesign.pulseshaping(8,'Raised Cosine','Ast,Beta',60,0.50);
   Hd = design(h);
   fvtool(Hd)
```

This code generates the following figure.
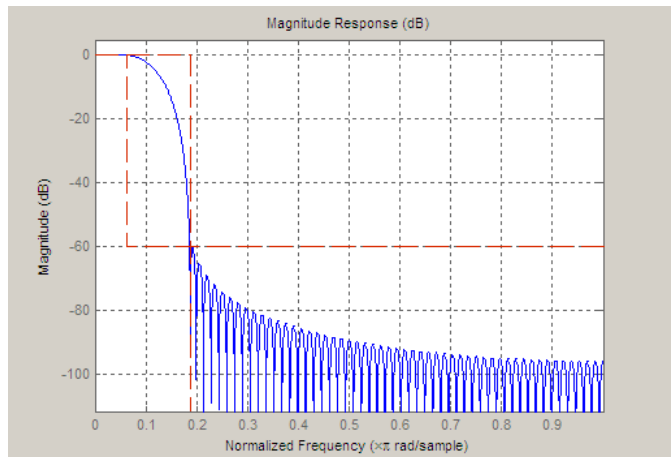
This example shows how to design a raised cosine filter that spans 8 symbol durations (i.e., of order 8 symbols), has a rolloff factor of 0.50, and oversampling factor of 10.

```
h  = fdesign.pulseshaping(10,'Raised Cosine','Nsym,Beta',8,0.50);
    Hd = design(h);
    fvtool(Hd, 'impulse')
```
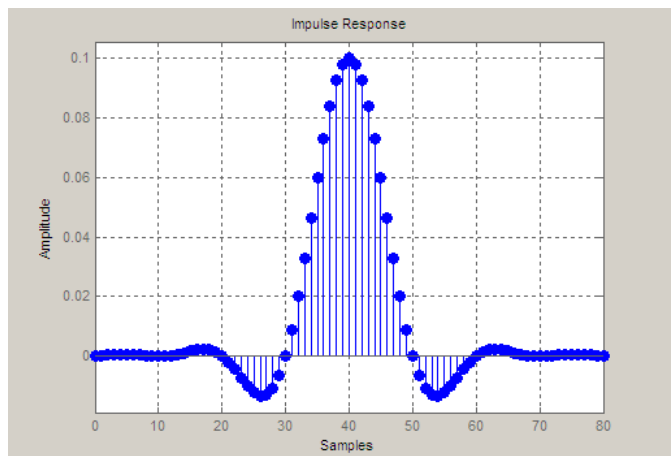
fdesign.pulseshaping

This example shows how to design a square root raised cosine filter of order 42, rolloff factor of 0.25, and 10 samples per symbol.

```
h  = fdesign.pulseshaping(10,'Square Root Raised Cosine','N,Beta',42);
  Hd = design(h);
  fvtool(Hd, 'impulse')
```
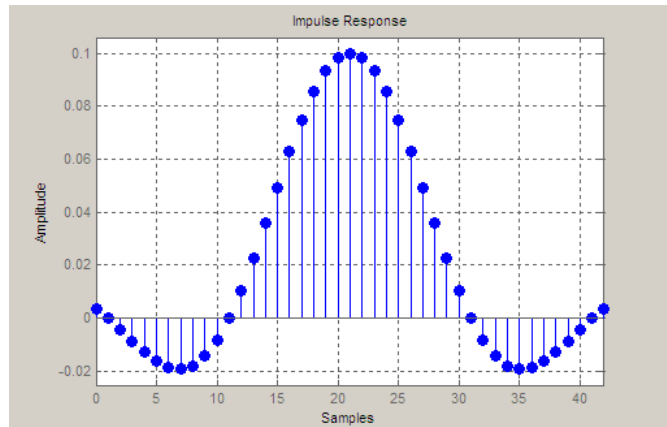

The following example demonstrates how to create a Gaussian pulse-shaping filter with an oversampling factor (sps) of 10, a bandwidth-time symbol product of 0.2, and 8 symbol periods. The sampling frequency is specified as 10 kHz.
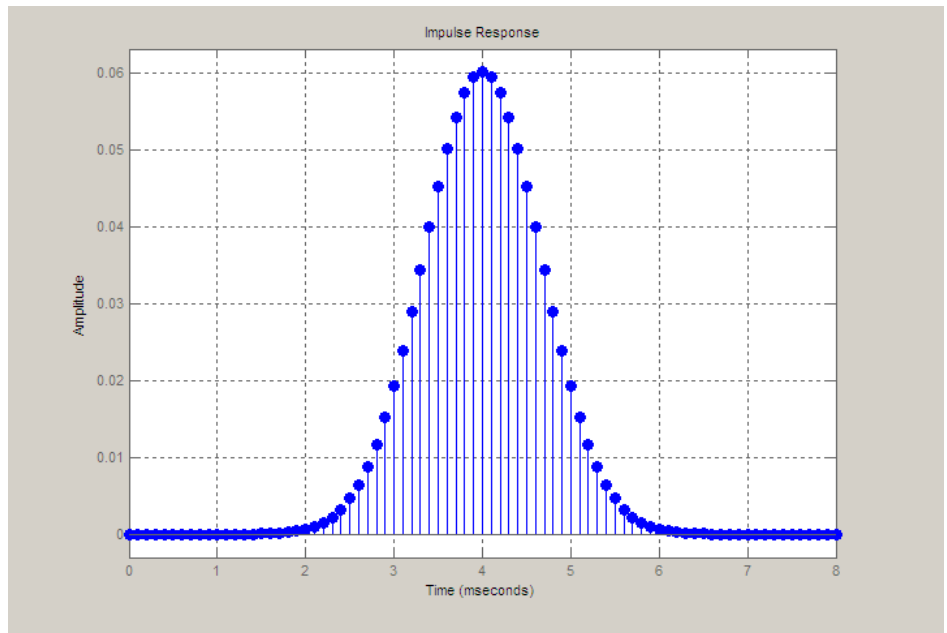
fdesign.rsrc

Purpose Rational-factor sample-rate converter specification

Syntax
```
D = fdesign.rsrc(L,M)
D = fdesign.rsrc(L,M,RESPONSE)
D = fdesign.rsrc(L,M,PULSESHAPINGRESPONSE,SPS)
D = fdesign.rsrc(L,M,CICRESPONSE,D)
D = fdesign.rsrc(L,M,RESPONSE,SPEC)
D = fdesign.rsrc(L,M,SPEC,specvalue1,specvalue2,...)
D = fdesign.rsrc(...,Fs)
D = fdesign.rsrc(...,MAGUNITS)
```

Description D = fdesign.rsrc(L,M) constructs a rational-factor sample-rate filter specification object D with the InterpolationFactor property equal to the positive integer L, the DecimationFactor property equal to the positive integer M and the Response property set to 'Nyquist'. The default values for the transition width and stopband attenuation in the Nyquist design are 0.1π radians/sample and 80 dB. If L is unspecified, L defaults to 3. If M is unspecified, M defaults to 2.

D = fdesign.rsrc(L,M,RESPONSE) constructs an rational-factor sample-rate converter with the interpolation factor L, decimation factor M, and the response you specify in RESPONSE.

D = fdesign.rsrc(L,M,PULSESHAPINGRESPONSE,SPS) constructs a pulse-shaping rational-factor sample-rate convertor filter specification object with the 'RESPONSE' property equal to one of the supported pulseshaping filters: 'Gaussian', 'Raised Cosine', or 'Square Root Raised Cosine'. SPS is the samples per symbol. The samples per symbol, SPS, must precede any specification string.

D = fdesign.rsrc(L,M,CICRESPONSE,D) constructs a CIC or CIC compensator rational-factor sample-rate convertor filter specification object with the 'RESPONSE' property equal to 'CIC' or 'CICCOMP'. D is the differential delay. The differential delay, D, must precede any specification string.

Because you are designing multirate filters, the specification strings available are not the same as the specifications for designing single-rate filters. The interpolation and decimation factors are not included in

the specification strings. Different filter responses support different specifications. The following table lists the supported response types and specification strings. The strings are not case sensitive.

Design String	Valid Specification Strings
`'Arbitrary Magnitude'`	See `fdesign.arbmag` for a description of the specification string entries. • `'N,F,A'` (default string) • `'N,B,F,A'`
`'Arbitrary Magnitude and Phase'`	See `fdesign.arbmagnphase` for a description of the specification string entries. • `'N,F,H'` (default string) • `'N,B,F,H'`
`'Bandpass'`	See `fdesign.bandpass` for a description of the specification string entries. • `'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'` (default string) • `'N,Fc1,Fc2'` • `'N,Fst1,Fp1,Fp2,Fst2'`
`'Bandstop'`	See `fdesign.bandstop` for a description of the specification string entries.
`'CIC'`	`'Fp,Fst,Ap,Ast'` — Only valid specification. `Fp` is the passband frequency, `Fst` is the stopband frequency, `Ap` is the passband ripple, and `Ast` is the stopband attenuation in decibels. To specify a CIC rational-factor sample-rate convertor, include the differential delay after `'CIC'` and before the filter specification string: `'Fp,Ast'`. For example: `d = fdesign.rsrc(2,2,'cic',4);`

Design String	Valid Specification Strings
`'CIC Compensator'`	See `fdesign.ciccomp` for a description of the specification string entries. • `'Fp,Fst,Ap,Ast'` (default string) • `'N,Fc,Ap,Ast'` • `'N,Fp,Ap,Ast'` • `'N,Fp,Fst'` • `'N,Fst,Ap,Ast'` To specify a CIC compensator rational-factor sample-rate convertor, include the differential delay after `'CICCOMP'` and before the filter specification string. For example: `d = fdesign.rsrc(2,2,'ciccomp',4);`
`'Differentiator'`	`'N'` — filter order
`'Gaussian'`	`'Nsym,BT` — `Nsym` is the filter order in symbols and `BT` is the bandwidth-symbol time product. The specification string must be preceded by an integer-valued `SamplesPerSymbol`.
`'Halfband`	See `fdesign.halfband` for a description of the specification string entries. • `'TW,Ast'` (default string) • `'N,TW'` • `` `N' `` • `'N,Ast'` If you use the quasi-linear IIR design method, `iirlinphase`, with a halfband specification, the interpolation factor must be 2.

Design String	Valid Specification Strings
`'Highpass'`	See `fdesign.highpass` for a description of the specification string entries. • `'Fst,Fp,Ast,Ap'` (default string) • `'N,F3db'` • `'N,Fc'` • `'N,Fc,Ast,Ap'` • `'N,Fp,Ast,Ap'` • `'N,Fst,Ast,Ap'` • `'N,Fst,Fp'` • `'N,Fst,Ast,Ap'` • `'N,Fst,Fp,Ast'`
`'Hilbert'`	See `fdesign.hilbert` for a description of the specification string entries. • `'N,TW'` (default string) • `'TW,Ap'`
`'Inverse-sinc Lowpass'`	See `fdesign.isinclp` for a description of the specification string entries. • `'Fp,Fst,Ap,Ast'` (default string)) • `'N,Fc,Ap,Ast'` • `'N,Fp,Fst'` • `'N,Fst,Ap,Ast'`

Design String	Valid Specification Strings
`'Inverse-sinc Highpass'`	See `fdesign.isinchp` for a description of the specification string entries. • `'Fst,Fp,Ast,Ap'` (default string)) • `'N,Fc,Ast,Ap'` • `'N,Fst,Fp'` • `'N,Fst,Ast,Ap'`
`'Lowpass'`	See `fdesign.lowpass` for a description of the specification string entries. • `'Fp,Fst,Ap,Ast'` (default string) • `'N,F3dB'` • `'N,Fc'` • `'N,Fc,Ap,Ast'` • `'N,Fp,Ap,Ast'` • `'N,Fp,Fst'` • `'N,Fp,Fst,Ap'` • `'N,Fp,Fst,Ast'` • `'N,Fst,Ap,Ast'`

Design String	Valid Specification Strings
`'Nyquist'`	See `fdesign.nyquist` for a description of the specification string entries. For all Nyquist specifications, you must specify the *L*th band. This typically corresponds to the interpolation factor so that the nonzero samples of the upsampler output are preserved. • `'TW,Ast'` (default string) • `'N'` • `'N,Ast'` • `'N,Ast'`
`'Raised Cosine'`	See `fdesign.pulseshaping` for a description of the specification string entries. • `'Ast,Beta'` (default string) • `'Nsym,Beta'` • `'N,Beta'` The specification string must be preceded by an integer-valued `SamplesPerSymbol`.
`'Square Root Raised Cosine'`	See `fdesign.pulseshaping` for a description of the specification string entries. • `'Ast,Beta'` (default string) • `'Nsym,Beta'` • `'N,Beta'` The specification string must be preceded by an integer-valued `SamplesPerSymbol`.

`D = fdesign.rsrc(L,M,RESPONSE,SPEC)` constructs object `D` and sets its `Specification` property to `SPEC`. Entries in the `SPEC` string represent various filter response features, such as the filter order, that govern

the filter design. Valid entries for SPEC depend on the design type of the specifications object.

When you add the SPEC input argument, you must also add the RESPONSE input argument.

D = fdesign.rsrc(L,M,SPEC,specvalue1,specvalue2,...) constructs an object D and sets its specifications at construction time.

D = fdesign.rsrc(...,Fs) provides the sampling frequency of the signal to be filtered. Fs must be specified as a scalar trailing the other numerical values provided. Fs is assumed to be in Hz as are all other frequency values provided.

D = fdesign.rsrc(...,MAGUNITS) specifies the units for any magnitude specification you provide in the input arguments. MAGUNITS can be one of

- 'linear' — specify the magnitude in linear units.

- 'dB' — specify the magnitude in dB (decibels).

- 'squared' — specify the magnitude in power units.

When you omit the MAGUNITS argument, fdesign assumes that all magnitudes are in decibels. Note that fdesign stores all magnitude specifications in decibels (converting to decibels when necessary) regardless of how you specify the magnitudes.

Examples   Design a rational-factor sample-rate converter. Set the rational sample-rate change to 5/3. Use the default Nyquist design with a transition width of 0.05π radians/sample and stopband attenuation of 40 dB. The *L*th band factor in the Nyquist design is equal to the interpolation factor.

```
d = fdesign.rsrc(5,3,'nyquist',5,.05,40);
hm = design(d,'kaiserwin'); %design with Kaiser window
```

Design a rational-factor sample-rate converter. Set the rational sample-rate change to 5/3. Use a Nyquist design with the 'N,TW'

specification string. Set the order equal to 12 and the transition width to 0.1π radians/sample. The *L*th band factor in the Nyquist design is equal to the interpolation factor.

```
d = fdesign.rsrc(5,3,'nyquist',5,'N,TW',12,0.1);
```

Design a rational-factor sample-rate converter. Assume the data are sampled at 10 kHz. Set the rational sample-rate change to 3/2. Use a Nyquist design with the 'N,TW' specification string. Set the order equal to 12 and the transition width to 100 Hz. The *L*th band factor in the Nyquist design is equal to the interpolation factor.

```
d = fdesign.rsrc(3,2,'nyquist',3,'N,TW',12,100,1e4);
hd = design(d,'equiripple');
```

See Also   design | designmethods | fdesign.rsrc | fdesign.interpolator | setspecs | fdesign.arbmag | fdesign.arbmagnphase

fftcoeffs

Purpose	Frequency-domain coefficients

Syntax
```
c = fftcoeffs(hd)
c = fftcoeffs(ha)
```

Description   c = fftcoeffs(hd) return the frequency-domain coefficients used when filtering with the dfilt.fftfir object. c contains the coefficients

c = fftcoeffs(ha) return the frequency-domain coefficients used when filtering with adaptfilt objects.

fftcoeffs applies to the following adaptive filter algorithms:

- adaptfilt.fdaf

- adaptfilt.pbfdaf

- adaptfilt.pbufdaf

- adaptfilt.ufdaf

Examples   This example demonstrates returning the FFT coefficients from the discrete-time filter hd.

```
b = [0.05 0.9 0.05];
len = 50;
hd = dfilt.fftfir(b,len);
c=fftcoeffs(hd);
```

Similarly, you can use fftcoeffs with the adaptive filters algorithms listed above. Start by constructing an adaptive filter ha.

```
d  = 16;                  % Number of samples of delay.
b = exp(1j*pi/4)*[-0.7 1]; % Numerator coefficients of channel.
a = [1 -0.7];             % Denominator coefficients of channel.
ntr= 1000;                % Number of iterations.
s  = sign(randn(1,ntr+d)) +...
1j*sign(randn(1,ntr+d));     % Baseband QPSK signal.
n = 0.1*(randn(1,ntr+d) + 1j*randn(1,ntr+d));  % Noise signal.
r = filter(b,a,s)+n;      % Received signal.
```

```
x   = r(1+d:ntr+d);          % Input signal (received signal).
s   = s(1:ntr);              % Desired signal (delayed QPSK signal).
del = 1;                     % Initial FFT input powers.
mu  = 0.1;                   % Step size.
lam = 0.9;                   % Averaging factor.
blocksize   = 8;                        % Block size.
ha = adaptfilt.pbufdaf(32,mu,1,del,lam,blocksize);
```

Here are the coefficients before you filter a signal.

```
c=fftcoeffs(ha);
% all coefficients are zero
NumNonzero =nnz(c);
```

Filtering a signal y produces complex nonzero coefficients that you use
fftcoeffs to see.

```
[y,e] = filter(ha,x,s);
c=fftcoeffs(ha);
```

See Also   adaptfilt.fdaf | adaptfilt.pbfdaf | adaptfilt.pbufdaf |
                   adaptfilt.ufdaf

filter

Purpose   Filter data with filter object

Syntax   **Fixed-Point Filter Syntaxes**
                `y = filter(hd,x)`
                `y = filter(hd,x,dim)`

                Adaptive Filter Syntax
                `y = filter(ha,x,d)`
                `[y,e] = filter(ha,x,d)`

                Multirate Filter Syntax
                `y = filter(hm,x)`
                `y = filter(hm,x,dim)`

Description   This reference page contains three sections that describe the syntaxes
                 for the filter objects:

- "Fixed-Point Filter Syntaxes" on page 7-640

- "Adaptive Filter Syntaxes" on page 7-641

- "Multirate Filter Syntaxes" on page 7-642

Fixed-Point Filter Syntaxes

`y = filter(hd,x)` filters a vector of real or complex input data x
through a fixed-point filter hd, producing filtered output data y. The
vectors x and y have the same length. filter stores the final conditions
for the filter in the States property of hd — hd.states.

When you set the property PersistentMemory to false (the default
setting), the initial conditions for the filter are set to zero before filtering
starts. To use nonzero initial conditions for hd, set PersistentMemory
to true. Then set hd.states to a vector of nstates(hd) elements, one
element for each state to set. If you specify a scalar for hd.states,
filter expands the scalar to a vector of the proper length for the states.
All elements of the expanded vector have the value of the scalar.

If x is a matrix, `y = filter(hd,x)` filters along each column of x to
produce a matrix y of independent channels. If x is a multidimensional

array, `y = filter(hd,x)` filters x along the first nonsingleton dimension of x.

To use nonzero initial conditions when you are filtering a matrix x, set the filter states to a matrix of initial condition values. Set the initial conditions by setting the `States` property for the filter (`hd.states`) to a matrix of `nstates(hd)` rows and `size(x,2)` columns.

`y = filter(hd,x,dim)` applies the filter hd to the input data located along the specific dimension of x specified by dim.

When you are filtering multichannel data, dim lets you specify which dimension of the input matrix to filter along — whether a row represents a channel or a column represents a channel. When you provide the dim input argument, the filter operates along the dimension specified by dim. When your input data x is a vector or matrix and dim is 1, each column of x is treated as a one input channel. When dim is 2, the filter treats each row of the input x as a channel.

To filter multichannel data in a loop environment, you must use the dim input argument to set the proper processing dimension.

You specify the initial conditions for each channel individually, when needed, by setting `hm.states` to a matrix of `nstates(hd)` rows (one row containing the states for one channel of input data) and `size(x,2)` columns (one column containing the filter states for each channel).

Adaptive Filter Syntaxes

`y = filter(ha,x,d)` filters a vector of real or complex input data x through an adaptive filter object ha, producing the estimated desired response data y from the process of adapting the filter. The vectors x and y have the same length. Use d for the desired signal. Note that d and x must be the same length signal chains.

`[y,e] = filter(ha,x,d)` produces the estimated desired response data y and the prediction error e (refer to previous syntax for more information).

filter

Multirate Filter Syntaxes

`y = filter(hd,x)` filters a vector of real or complex input data `x` through a fixed-point filter `hd`, producing filtered output data `y`. The vectors `x` and `y` have the same length. filter stores the final conditions for the filter in the `States` property of `hd` — `hd.states`.

`y = filter(hm,x,dim)` applies the filter `hd` to the input data located along the specific dimension of `x` specified by `dim`.

When you are filtering multichannel data, `dim` lets you specify which dimension of the input matrix to filter along — whether a row represents a channel or a column represents a channel. When you provide the `dim` input argument, the filter operates along the dimension specified by `dim`. When your input data `x` is a vector or matrix and `dim` is 1, each column of `x` is treated as a one input channel. When `dim` is 2, the filter treats each row of the input `x` as a channel.

To filter multichannel data in a loop environment, you must use the `dim` input argument to set the processing dimension.

You specify the initial conditions for each channel individually, when needed, by setting `hm.states` to a matrix of `nstates(hm)` rows (one row containing the states for one channel of input data) and `size(x,2)` columns (one column containing the filter states for each channel).

The number of data samples in your input data set `x` does not need to be a multiple of the rate change factor `r` for the object. When the rate change factor is not an even divisor of the number of input samples `x`, `filter` processes the samples as shown in the following figure, where the rate change factor is 3 and the number of input samples is 23. Decimators always take the first input sample to generate the first output sample. After that, the next output sample comes after each `r` number of input samples.

Given 23 Input Data Samples

You Get 7 Output Data Samples
After Decimation By 3

2 Processed Samples
That Did Not Generate
An Output Sample
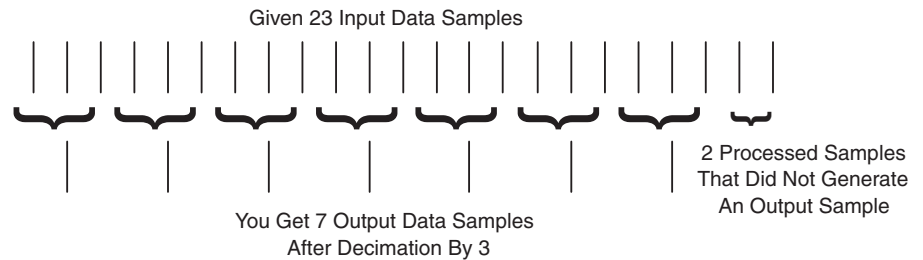
Examples   Filter a signal using a filter with various initial conditions (IC) or no
initial conditions.

```
x = randn(100,1);    % Original signal.
b = fir1(50,.4);     % 50th-order linear-phase FIR filter.
hd = dfilt.dffir(b);    % Direct-form FIR implementation.

% Do not set specific initial conditions.

y1 = filter(hd,x);   % 'PersistentMemory'='false'(default).
zf = hd.states;      % Final conditions.
```

Now use nonzero initial conditions by setting ICs after before you filter.

```
hd.persistentmemory = true;
hd.states = 1;       % Uses scalar expansion.
y2 = filter(hd,x);
stem([y1 y2])        % Different sequences at beginning.
```

Looking at the stem plot shows that the sequences are different at the
beginning of the filter process.
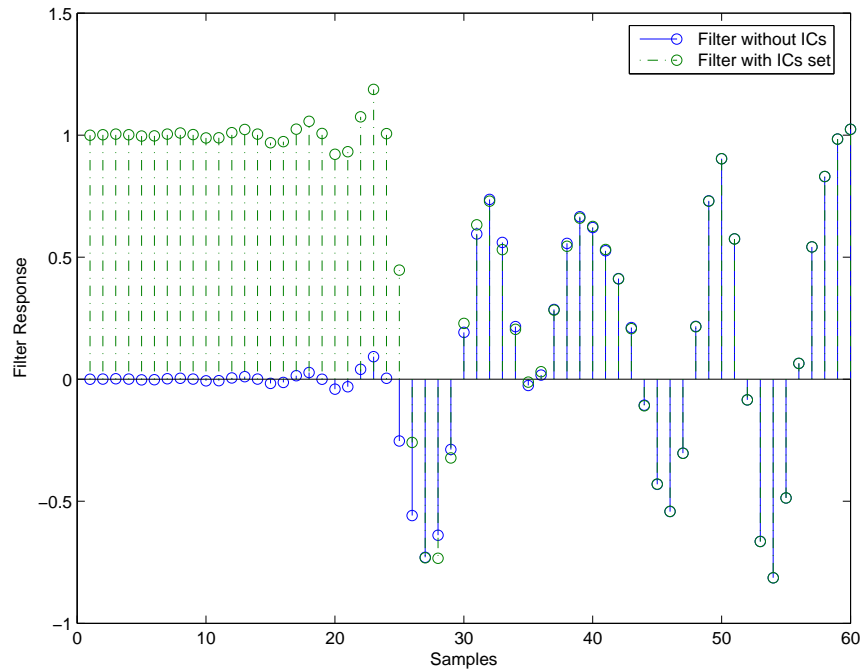
Here is one way to use `filter` with streaming data.

```
reset(hd);           % Clear filter history.
y3 = filter(hd,x);   % Filter entire signal in one block.
```

As an experiment, repeat the process, filtering the data as sections, rather than in streaming form.

```
reset(hd);              % Clear filter history.
yloop = zeros(20,5);    % Preallocate output array.
xblock = reshape(x,[20 5]);
for i=1:5,
   yloop(:,i) = filter(hd,xblock(:,i));
end
```

Use a stem plot to see the comparison between streaming and block-by-block filtering.

```
stem([y3 yloop(:)]);
```
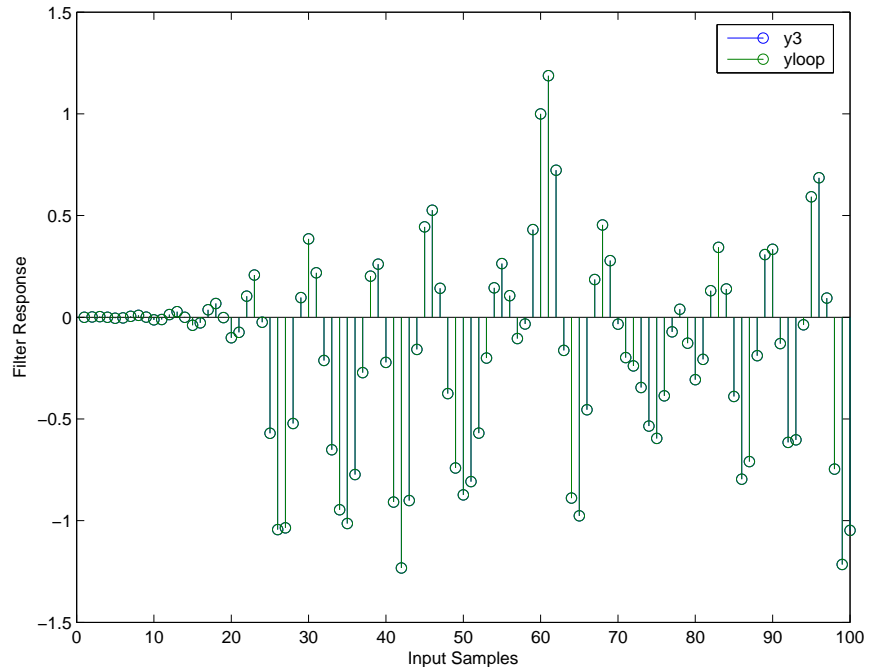

Filtering the signal section-by-section is equivalent to filtering the entire signal at once.

To show the similarity between filtering with discrete-time and with multirate filters, this example demonstrates multirate filtering.

```
Fs = 44.1e3;             % Original sampling frequency: 44.1kHz.
n = [0:10239].';         % 10240 samples, 0.232 second long signal.
x  = sin(2*pi*1e3/Fs*n); % Original signal, sinusoid at 1kHz.
m = 2;                   % Decimation factor.
hm = mfilt.firdecim(m);  % Use the default filter.
```

First, filter without setting initial conditions.

```
y1 = filter(hm,x);        % PersistentMemory is false (default).
zf = hm.states;           % Final conditions.
```

This time, set nonzero initial conditions before filtering the data.

```
hm.persistentmemory = true;
hm.states = 1;          % Uses scalar expansion to set ICs.
y2 = filter(hm,x);
stem([y1(1:60) y2(1:60)]) % Show the filtering results.
```

Note the different sequences at the start of filtering.

Finally, try filtering streaming data.

```
reset(hm);             % Clear the filter history.
y3 = filter(hm,x);     % Filter entire signal in one block.
```

As with the discrete-time filter, filtering the signal section by section is equivalent to filtering the entire signal at once.

```
reset(hm);              % Clear filter history again.
yloop = zeros(1024,5);  % Preallocate output array.
xblock = reshape(x,[2048 5]);
for i=1:5,
   yloop(:,i) = filter(hm,xblock(:,i));
end
stem([y3 yloop(:)]);
```

Algorithms   Quantized Filters

The `filter` command implements fixed- or floating-point arithmetic on the quantized filter structure you specify.

The algorithm applied by `filter` when you use a discrete-time filter object on an input signal depends on the response you chose for the filter, such as lowpass or Nyquist or bandstop. To learn more about each filter algorithm, refer to the literature reference provided on the appropriate discrete-time filter reference page.

> **Note** dfilt/filter does not normalize the filter coefficients automatically. Function filter supplied by MATLAB does normalize the coefficients.

Adaptive Filters

The algorithm used by filter when you apply an adaptive filter object to a signal depends on the algorithm you chose for your adaptive filter. To learn more about each adaptive filter algorithm, refer to the literature reference provided on the appropriate adaptfilt.*algorithm* reference page.

Multirate Filters

The algorithm applied by filter when you apply a multirate filter objects to signals depends on the algorithm you chose for the filter — the form of the multirate filter, such as decimator or interpolator. To learn more about each filter algorithm, refer to the literature reference provided on the appropriate multirate filter reference page.

References [1] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989.

See Also adaptfilt | impz | mfilt | nstates | dfilt

filterbuilder

Purpose   GUI-based filter design

Syntax   `filterbuilder(h)`
                  `filterbuilder('`*response*`')`

Description   `filterbuilder` starts a GUI-based tool for building filters. It relies
                  on the `fdesign` object-object oriented filter design paradigm, and is
                  intended to reduce development time during the filter design process.
                  `filterbuilder` uses a specification-centered approach to find the best
                  algorithm for the desired response.

Note You must have the Signal Processing Toolbox installed to use
`fdesign` and `filterbuilder`. Some of the features described below
may be unavailable if your installation does not additionally include the
DSP System Toolbox. You can verify the presence of both toolboxes by
typing `ver` at the command prompt.

The `filterbuilder` GUI contains many features not available in
FDATool. For more information on how to use `filterbuilder`, see
"Filterbuilder Design Process".

To use `filterbuilder`, enter `filterbuilder` at the MATLAB command
line using one of three approaches:

- Simply enter `filterbuilder`. MATLAB opens a dialog for you to
  select a filter response type. After you select a filter response type,
  `filterbuilder` launches the appropriate filter design dialog box.

- Enter `filterbuilder(h)`, where h is an existing filter object. For
  example, if h is a bandpass filter, `filterbuilder(h)` opens the
  bandpass filter design dialog box. (The h object must have been
  created using `filterbuilder` or must be a `dfilt` ,`mfilt`, or filter
  System object created using fdesign.)

Note You must have the DSP System Toolbox software to create and import filter System objects.

- Enter filterbuilder('*response*'), replacing *response* with a response string from the following table. MATLAB opens a filter design dialog that corresponds to the response string.

Note You must have the DSP System Toolbox software to implement a number of the filter designs listed in the following table. If you only have the Signal Processing Toolbox software, you can design a limited set of the following filter-response types.

Response String	Description of Resulting Filter Design	Filter Object
arbgrpdelay	Arbitrary group delay filter design	fdesign.arbgrpdelay
arbmag	Arbitrary magnitude filter design	fdesign.arbmag
arbmagnphase	Arbitrary response filter (magnitude and phase)	fdesign.arbmagnphase
audioweighting	Audio weighting filter	fdesign.audioweighting
bandpass or bp	Bandpass filter	fdesign.bandpass
bandstop or bs	Bandstop filter	fdesign.bandstop
cic	CIC filter	fdesign.decimator(M,'cic',... or
ciccomp	CIC compensator	fdesign.ciccomp r(L,'cic',
comb	Comb filter	fdesign.comb

and
fdesign.interpolator

Response String	Description of Resulting Filter Design	Filter Object
diff	Differentiator filter	`fdesign.differentiator`
fracdelay	Fractional delay filter	`fdesign.fracdelay`
halfband or `hb`	Halfband filter	`fdesign.halfband`
highpass or `hp`	Highpass filter	`fdesign.highpass`
hilb	Hilbert filter	`fdesign.hilbert`
isinc, isinclp, or isinchp	Inverse sinc lowpass or highpass filter	`fdesign.isinclp` and `fdesign.isinchp`
lowpass or `lp`	Lowpass filter (default)	`fdesign.lowpass`
notch	Notch filter	`fdesign.notch`
nyquist	Nyquist filter	`fdesign.nyquist`
octave	Octave filter	`fdesign.octave`
parameq	Parametric equalizer filter	`fdesign.parameq`
peak	Peak filter	`fdesign.peak`
pulseshaping	Pulse-shaping filter	`fdesign.pulseshaping`

Note Because they do not change the filter structure, the magnitude specifications and design method are tunable when using `filterbuilder`.

Filterbuilder Design Panes

Main Design Pane

The main pane of filterbuilder varies depending on the filter response type, but the basic structure is the same. The following figure shows the basic layout of the dialog box.
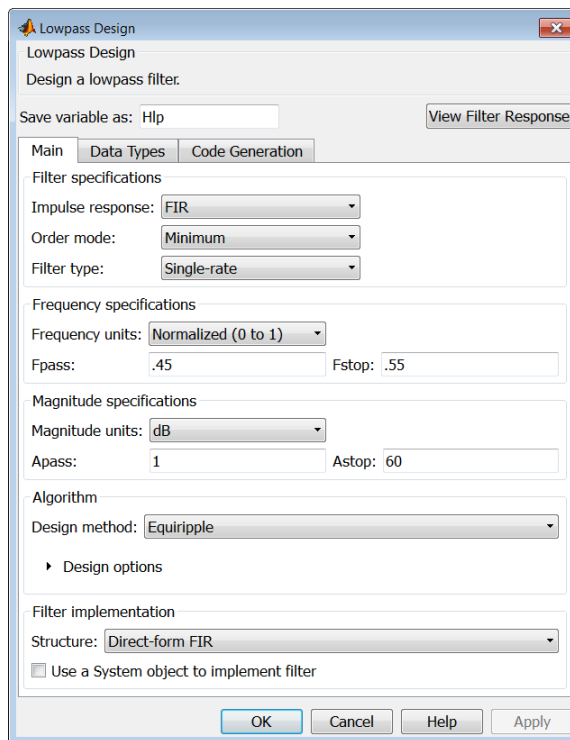
As you choose the response for the filter, the available options and design parameters displayed in the dialog box change. This display allows you to focus only on parameters that make sense in the context of your filter design.

Every filter design dialog box includes the options displayed at the top of the dialog box, shown in the following figure.

- **Save variable as** — When you click **Apply** to apply your changes or **OK** to close this dialog box, filterbuilder saves the current filter to your MATLAB workspace as a filter object with the name you enter.

- **View Filter Response** — Displays the magnitude response for the current filter specifications and design method by opening the Filter Visualization Tool (fvtool).

Note The filterbuilder dialog box includes an **Apply** option. Each time you click **Apply**, filterbuilder writes the modified filter to your MATLAB workspace. This modified filter has the variable name you assign in **Save variable as**. To apply changes without overwriting the variable in you workspace, change the variable name in **Save variable as** before you click **Apply**.

There are three tabs in the Filterbuilder dialog box, containing three panes: **Main**, **Data Types**, and **Code Generation**. The first pane changes according to the filter being designed. The last two panes are the same for all filters. These panes are discussed in the following sections.

Data Types Pane

The second tab in the Filterbuilder dialog box is shown in the following figure.
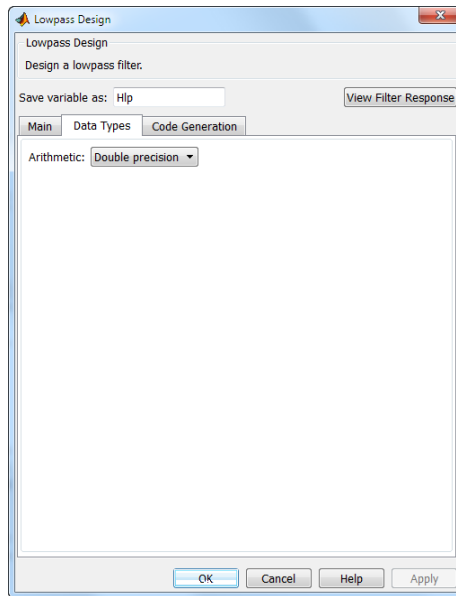
The **Arithmetic** drop down box allows the choice of `Double precision`, `Single precision`, or `Fixed point`. Some of these options may be unavailable depending on the filter parameters. The following table describes these options.

Arithmetic List Entry	Effect on the Filter
`Double precision`	All filtering operations and coefficients use double-precision, floating-point representations and math. When you use `filterbuilder` to

Arithmetic List Entry	Effect on the Filter
	create a filter, `double precision` is the default value for the Arithmetic property.
`Single precision`	All filtering operations and coefficients use single-precision floating-point representations and math.
`Fixed point`	This string applies selected default values, typically used on many digital processors, for the properties in the fixed-point filter. These properties include coefficient word lengths, fraction lengths, and various operating modes. This setting allows signed fixed data types only. Fixed-point filter design with `filterbuilder` is available only when you install Fixed-Point Toolbox software along with DSP System Toolbox software.

The following figure shows the **Data Types** pane after you select `Fixed point` for **Arithmetic** and set **Filter internals** to `Specify precision`. This figure shows the **Data Types** pane for the case where the **Use a System object to implement filter** check box is not selected in the **Main** pane.

Input signal

Specify the format the filter applies to data to be filtered. For all cases, `filterbuilder` implements filters that use binary point scaling and signed input. You set the word length and fraction length as needed.

Coefficients

Choose how you specify the word length and the fraction length of the filter numerator and denominator coefficients:

- `Specify word length` enables you to enter the word length of the coefficients in bits. In this mode, `filterbuilder` automatically sets the fraction length of the coefficients to the binary-point only scaling that provides the best possible precision for the value and word length of the coefficients.

- `Binary point scaling` enables you to enter the word length and the fraction length of the coefficients in bits. If applicable, enter separate fraction lengths for the numerator and denominator coefficients.

- The filter coefficients do not obey the **Rounding mode** and **Overflow mode** parameters that are available when you select `Specify precision` from the Filter internals list. Coefficients are always saturated and rounded to `Nearest`.

Section Input

Choose how you specify the word length and the fraction length of the fixed-point data type going into each section of an SOS filter. This parameter is visible only when the selected filter structure is IIR and SOS.

- `Binary point scaling` enables you to enter the word and fraction lengths of the section input in bits.

- `Specify word length` enables you to enter the word lengths in bits.

Section Output

Choose how you specify the word length and the fraction length of the fixed-point data type coming out of each section of an SOS

filter. This parameter is visible only when the selected filter structure is IIR and SOS.

- `Binary point scaling` enables you to enter the word and fraction lengths of the section output in bits.

- `Specify word length` enables you to enter the output word lengths in bits.

State

Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Use this parameter to specify how to designate the state word and fraction lengths. This parameter is not visible for direct form and direct form I filter structures because `filterbuilder` deduces the state directly from the input format. States always use signed representation:

- `Binary point scaling` enables you to enter the word length and the fraction length of the accumulator in bits.

- `Specify precision` enables you to enter the word length and fraction length in bits (if available).

Product

Determines how the filter handles the output of product operations. Choose from the following options:

- `Full precision` — Maintain full precision in the result.

- `Keep LSB` — Keep the least significant bit in the result when you need to shorten the data words.

- `Specify Precision` — Enables you to set the precision (the fraction length) used by the output from the multiplies.

Filter internals

Specify how the fixed-point filter performs arithmetic operations within the filter. The affected filter portions are filter products, sums, states, and output. Select one of these options:

- Full precision — Specifies that the filter maintains full precision in all calculations for products, output, and in the accumulator.

- Specify precision — Set the word and fraction lengths applied to the results of product operations, the filter output, and the accumulator. Selecting this option enables the word and fraction length controls.

Signed

Selecting this option directs the filter to use signed representations for the filter coefficients.

Word length

Sets the word length for the associated filter parameter in bits.

Fraction length

Sets the fraction length for the associate filter parameter in bits.

Accum

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths.

Determines how the accumulator outputs stored values. Choose from the following options:

- Full precision — Maintain full precision in the accumulator.

- Keep MSB — Keep the most significant bit in the accumulator.

- Keep LSB — Keep the least significant bit in the accumulator when you need to shorten the data words.

- Specify Precision — Enables you to set the precision (the fraction length) used by the accumulator.

Output

Sets the mode the filter uses to scale the output data after filtering. You have the following choices:

- Avoid Overflow — Set the output data fraction length to avoid causing the data to overflow. Avoid overflow is considered

the conservative setting because it is independent of the input data values and range.

- `Best Precision` — Set the output data fraction length to maximize the precision in the output data.

- `Specify Precision` — Set the fraction length used by the filtered data.

Fixed-point operational parameters

Parameters in this group control how the filter rounds fixed-point values and how it treats values that overflow.

Rounding mode

Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths).

- `ceil` - Round toward positive infinity.

- `convergent` - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.

- `zero/fix` - Round toward zero.

- `floor` - Round toward negative infinity.

- `nearest` - Round toward nearest. Ties round toward positive infinity.

- `round` - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

Overflow mode

Sets the mode the filter uses to respond to overflow conditions in fixed-point arithmetic. Choose from the following options:

- Saturate — Limit the output to the largest positive or negative representable value.

- Wrap — Set overflowing values to the nearest representable value using modular arithmetic.

The choice you make affects everything except coefficient values and input data which always round. In most cases, products do not overflow—they maintain full precision.

Cast before sum

Specifies whether to cast numeric data to the appropriate accumulator format before performing sum operations. Selecting **Cast before sum** ensures that the results of the affected sum operations match most closely the results found on most digital signal processors. Performing the cast operation before the summation adds one or two additional quantization operations that can add error sources to your filter results.

If you clear **Cast before sum**, the filter prevents the addends from being cast to the sum format before the addition operation. Choose this setting to get the most accurate results from summations without considering the hardware your filter might use. The input format referenced by **Cast before sum** depends on the filter structure you are using.

The effect of clearing or selecting **Cast before sum** is as follows:

- Cleared — Configures filter summation operations to retain the addends in the format carried from the previous operation.

- Selected — Configures filter summation operations to convert the input format of the addends to match the summation output format before performing the summation operation. Usually, selecting **Cast before sum** generates results from the summation that more closely match those found from digital signal processors.

Code Generation Pane

The code generation pane contains options for various implementations of the completed filter design. Depending on your installation, you can generate MATLAB, VHDL, and Verilog code from the designed filter. You can also choose to create or update a Simulink model from the designed filter. The following section explains these options.
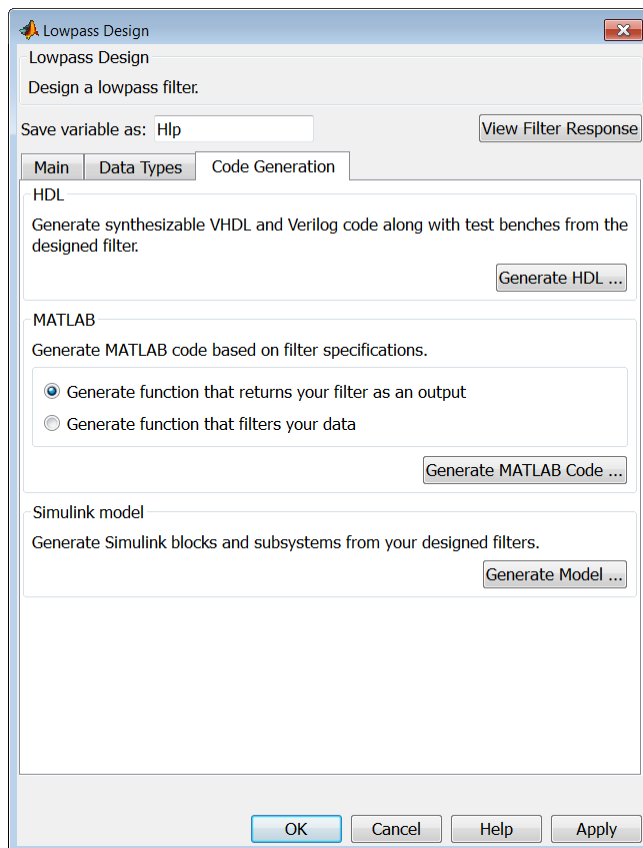
filterbuilder

HDL

For more information on this option, see "Opening the Filter Design HDL Coder™ GUI From the filterbuilder GUI".

MATLAB

Generate MATLAB code based on filter specifications

- **Generate function that returns your filter as an output**

  Selecting this option generates a function that designs either a DFILT/MFILT object or a system object (depending on whether you have selected the **Use a System object to implement the filter** check box) using fdesign. The function call returns a filter object.

- **Generate function that filters your data**

  Selecting this option generates a function that takes data as input, and outputs data filtered with the designed filter.

  Clicking on the **Generate MATLAB code** button, brings up a Save File dialog. Specify the file name and location, and save. The filter is now contained in an editable file.

Simulink Model

Generate Simulink blocks and subsystems from your designed filters

When the **Use a System object to implement filter** check box is selected in the **Main** pane, you are able to generate Simulink models as long as the **Arithmetic** is not set to Fixed point in the **Data Types** pane. If the **Arithmetic** is set to Fixed point, the **Generate Model** button in the **Simulink model** panel will be disabled.

Clicking on the **Generate Model** button brings up the **Export to Simulink** dialog box, as shown in the following figure.
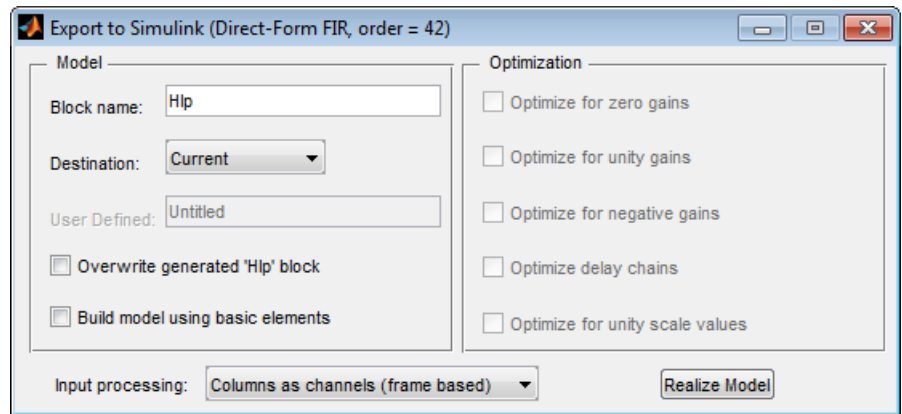
You can set the following parameters in this dialog box:

- **Block Name** — The name for the new subsystem block, set to **Filter** by default.

- **Destination** — **Current** saves the generated model to the current Simulink model; **New** creates a new model to contain the generated block; **User Defined** creates a new model or subsystem to the user-specified location enumerated in the **User Defined** text box.

- **Overwrite generated 'Filter' block** — When this check box is selected, DSP System Toolbox software overwrites an existing block with the name specified in **Block Name**; when cleared, creates a new block with the same name.

- **Build model using basic elements** — When this check box is selected, DSP System Toolbox software builds the model using only basic blocks.

- **Optimize for zero gains** — When this check box is selected, DSP System Toolbox software removes all zero gain blocks from the model.

- **Optimize for unity gains** — When this check box is selected, DSP System Toolbox software replaces all unity gains with direct connections.

- **Optimize for negative gains** — When this check box is selected, DSP System Toolbox software removes all negative unity gain blocks, and changes sign at the nearest summation block.

- **Optimize delay chains** — When this check box is selected, DSP System Toolbox software replaces delay chains made up of *n* unit delays with a single delay by *n*.

- **Optimize for unity scale values** — When this check box is selected, DSP System Toolbox software removes all scale value multiplications by 1 from the filter structure.

- **Input processing** — Specify how the generated filter block or subsystem block processes the input. Depending on the type of filter you are designing, one or both of the following options may be available:

  - `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

  - `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

  For more information about sample- and frame-based processing, see "Sample- and Frame-Based Concepts".

- **Realize Model** — DSP System Toolbox software builds the model with the set parameters.

Filter Responses

Select your filter response from the `filterbuilder` **Response Selection** main menu.

If you have the DSP System Toolbox software, the following **Response Selection** menu appears.
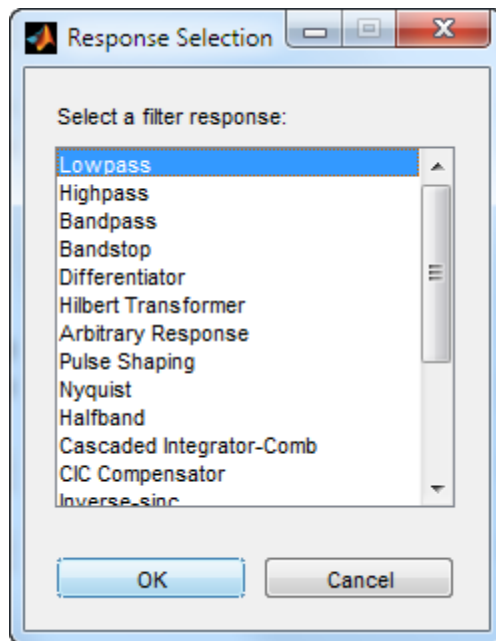
Select your desired filter response from the menu and design your filter.

The following sections describe the options available for each response type.

Arbitrary Response Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

This dialog only applies if you have the DSP System Toolbox software. Select either FIR or IIR from the drop down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly. Arbitrary group delay designs are only available if **Impulse response** is IIR. Without the DSP System Toolbox, the only available arbitrary response filter design is FIR.

Order mode

This dialog only applies if you have the DSP System Toolbox software. Choose Minimum or Specify. Choosing Specify enables the **Order** dialog.

Order

This dialog only applies when **Order mode** is Specify. For an FIR design, specify the filter order. For an IIR design, you can specify an equal order for the numerator and denominator, or you can specify different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box. Because the Signal Processing Toolbox only supports FIR arbitrary-magnitude filters, you do not have the option to specify a denominator order.

Denominator order

Select the check box and enter the denominator order. This option is enabled only if IIR is selected for **Impulse response**.

Filter type

This dialog only applies if you have the DSP System Toolbox software and is only available for FIR filters. Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods

and structures that are available to implement your filter. By default, `filterbuilder` specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2 for `Decimator` and 3 for `Sample-rate converter`.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Response Specification

Number of Bands

Select the number of bands in the filter. Multiband design is available for both FIR and IIR filters.

Specify response as:

Specify the response as `Amplitudes`, `Magnitudes and phase`, `Frequency response`, or `Group delay`. `Amplitudes` is the only option if you do not have the DSP System Toolbox software. `Group delay` is only available for IIR designs.

Frequency units

Specify frequency units as either `Normalized`, `Hz`, `kHz`, `MHz`, or `GHz`.

Input Fs

Enter the input sampling frequency in the units specified in the **Frequency units** drop-down box. This option is enabled when **Frequency units** is set to an option in hertz.

Band Properties

These properties are modified automatically depending on the response chosen in the **Specify response as** drop-down box. Two or three columns are presented for input. The first column is always Frequencies. The other columns are either Amplitudes, Magnitudes, Phases, or Frequency Response. Enter the corresponding vectors of values for each column.

- **Frequencies** and **Amplitudes** — These columns are presented for input if you select Amplitudes in the **Specify response as** drop-down box.

- **Frequencies**, **Magnitudes**, and **Phases** — These columns are presented for input if the response chosen in the **Specify response as** drop-down box is Magnitudes and phases.

- **Frequencies** and **Frequency response** — These columns are presented for input if the response chosen in the **Specify response as** drop-down box is Frequency response.

Algorithm

The options for each design are specific for each design method. In the arbitrary response design, the available options also depend on the **Response specifications**. This section does not present all of the available options for all designs and design methods.

Design Method

Select the design method for the filter. Different methods are enabled depending on the defining parameters entered in the previous sections.

Design Options

- **Window** — Valid when the **Design method** is `Frequency Sampling`. Replace the square brackets with the name of a `window` function or function handle. For example, `'hamming'` or `@hamming`. If the window function takes parameters other than the length, use a cell array. For example, {`` `kaiser' ``,`3.5`} or {`@chebwin,60`}.

- **Density factor** — Valid when the **Design method** is `equiripple`. Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

  Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

  The default changes to 20 for an IIR arbitrary group delay design.

- **Phase constraint** — Valid when the **Design method** is `equiripple`, you have the DSP System Toolbox installed, and **Specify response as** is set to `Amplitudes`. Choose one of `Linear`, `Minimum`, or `Maximum`.

- **Weights** — Uses the weights in **Weights** to weight the error for a single-band design. If you have multiple frequency bands, the **Weights** design option changes to **B1 Weights**, **B2 Weights** to designate the separate bands. Use **Bi Weights** to specify weights for the i-th band. The **Bi Weights** design option is only available when you specify the i-th band as an unconstrained.

- **Bi forced frequency point** — This option is only available in a multi-band constrained equiripple design when **Specify response as** is `Amplitudes`. **Bi forced frequency point** is

the frequency point in the i-th band at which the response is forced to be zero. The index **i** corresponds to the frequency bands in **Band properties**. For example, if you specify two bands in **Band properties**, you have **B1 forced frequency point** and **B2 forced frequency point**.

- **Norm** — Valid only for IIR arbitrary group delay designs. **Norm** is the norm used in the optimization. The default value is 128, which essentially equals the L-infinity norm. The norm must be even.

- **Max pole radius** — Valid only for IIR arbitrary group delay designs. Constrains the maximum pole radius. The default is 0.999999. Reducing the **Max pole radius** can produce a transfer function more resistant to quantization.

- **Init norm** — Valid only for IIR arbitrary group delay designs. The initial norm used in the optimization. The default initial norm is 2.

- **Init numerator** — Specifies an initial estimate of the filter numerator coefficients.

- **Init denominator** — Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems. In allpass filters, you only have to specify either the denominator or numerator coefficients. If you specify the denominator coefficients, you can obtain the numerator coefficients.

Filter implementation

Structure

Select the structure for the filter. The available filter structures depend on the parameters you select for your filter.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned

off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Audio Weighting Filter Design Dialog Box — Main Pane
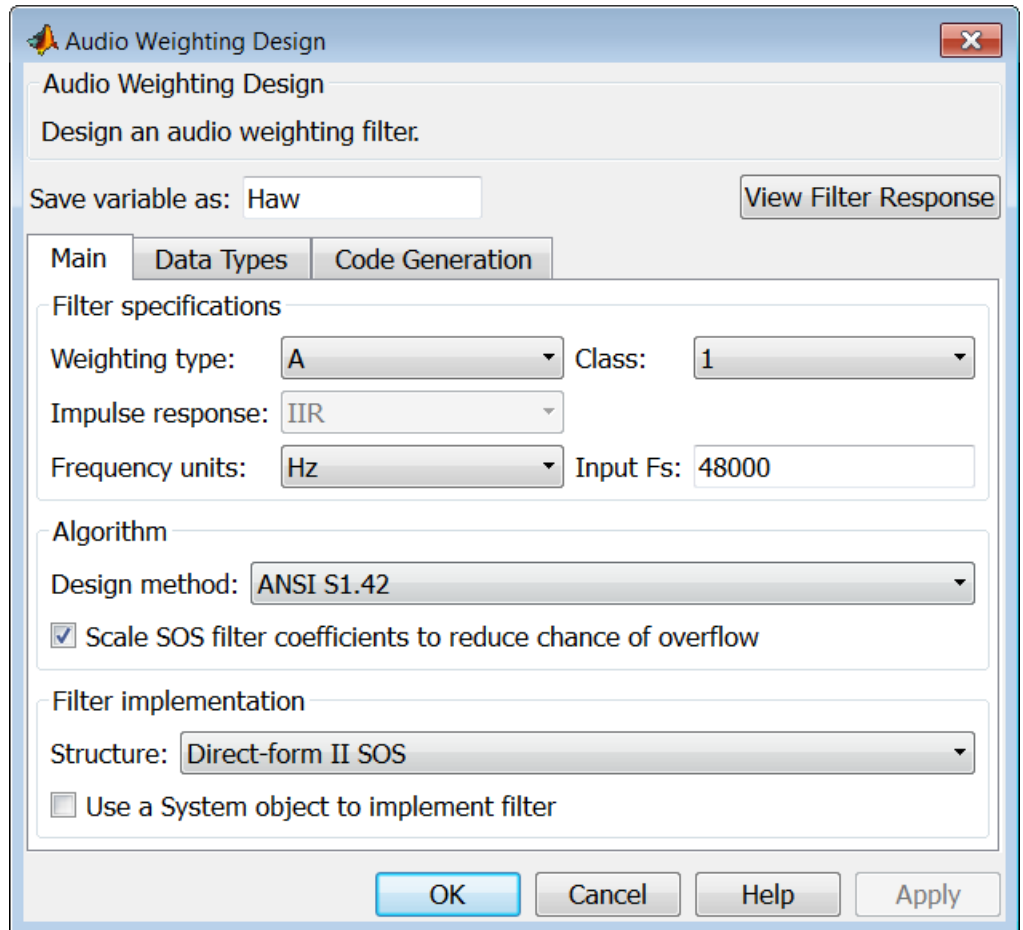
Filter specifications

- **Weighting type** — The weighting type defines the frequency response of the filter. The valid weighting types are: A, C , C-message, ITU-T 0.41, and ITU-R 468–4 weighting. See `fdesign.audioweighting` for definitions of the weighting types.

- **Class** — Filter class is only applicable for A weighting and C weighting filters. The filter class describes the frequency-dependent tolerances specified in the relevant standards. There are two possible class values: 1 and 2. Class 1 weighting filters have stricter tolerances than class 2 filters. The filter class value does not affect the design. The class value is only used to provide a specification mask in `fvtool` for the analysis of the filter design.

- **Impulse response** — Impulse response type as one of `IIR` or `FIR`. For A, C , C-message, and ITU-R 468–4 filter, IIR is the only option. For a ITU-T 0.41 weighting filter, FIR is the only option.

- **Frequency units** — Choose `Hz`, `kHz`, `MHz`, or `GHz`. Normalized frequency designs are not supported for audio weighting filters.

- **Input Fs** — The sampling frequency in **Frequency units**. For example, if **Frequency units** is set to `kHz`, setting **Input Fs** to 40 is equivalent to a 40 kHz sampling frequency.

Algorithm

- **Design method** — Valid design methods depend on the weighting type. For type A and C weighting filters, the only valid design type is `ANSI S1.42`. This is an IIR design method that follows ANSI standard S1.42–2001. For a C message filter, the only valid design method is `Bell 41009`, which is an IIR design method following the Bell System Technical Reference PUB 41009. For a ITU-R 468–4 weighting filter, you can design an IIR or FIR filter. If you choose an IIR design, the design method is `IIR least p-norm`. If you choose an FIR design, the design method choices are: `Equirriple` or `Frequency Sampling`. For an ITU-T 0.41 weighting filter, the available FIR design methods are `equirriple` or `Frequency Sampling`

- **Scale SOS filter coefficients to reduce chance of overflow** — Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Filter implementation

- **Structure** — For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. For audio weighting IIR filter designs, you can choose direct form I or II biquad (SOS). You can also choose to implement these structures in transposed form.

  For FIR designs, you can choose direct form, direct-form transposed, direct-form symmetric, direct-form asymmetric structures, or an overlap and add structure.

- **Use a System object to implement filter** — Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Bandpass Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select Minimum (the default) or Specify from the drop-down box. Selecting Specify enables the **Order** option so you can enter the filter order.

If you have the DSP System Toolbox software installed, you can specify IIR filters with different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type — This dialog only applies if you have the DSP System Toolbox software.

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

Order

> Enter the filter order. This option is enabled only if you select
> `Specify` for **Order mode**.

Decimation Factor

> Enter the decimation factor. This option is enabled only if the
> **Filter type** is set to `Decimator` or `Sample-rate converter`. The
> default factor value is 2.

Interpolation Factor

> Enter the decimation factor. This option is enabled only if the
> **Filter type** is set to `Interpolator` or `Sample-rate converter`.
> The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response
curve. Graphically, the filter specifications look similar to those shown
in the following figure.

In the figure, regions between specification values such as Fstop1 and
Fpass1 represent transition regions where the filter response is not
explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. This dialog applies only when **Order mode** is **Specify**.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.

- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.

- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.

- `3dB points` — Define the filter response by specifying the locations of the 3 dB points (IIR filters). The 3-dB point is the frequency for the point 3 dB below the passband value.

- `3dB points and passband width` — Define the filter by specifying frequencies for the 3-dB points in the filter response and the width of the passband. (IIR filters)

- `3dB points and stopband widths` — Define the filter by specifying frequencies for the 3-dB points in the filter response and the width of the stopband. (IIR filters)

- `6dB points` — Define the filter response by specifying the locations of the 6-dB points. The 6-dB point is the frequency for the point 6dB below the passband value. (FIR filters)

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in hertz, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fstop1

Enter the frequency at the edge of the end of the first stopband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fpass1

Enter the frequency at the edge of the start of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fpass2

Enter the frequency at the edge of the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop2

Enter the frequency at the edge of the start of the second stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude constraints

Specify as Unconstrained or Constrained bands. You must have the DSP System Toolbox software to select Constrained bands. Selecting Constrained bands enables dialogs for both stopbands and the passband: **Astop1**, **Astop2**, and **Apass**. You cannot specify constraints for all three bands simultaneously.

Setting **Magnitude constraints** to `Constrained bands` enables the **Wstop** and **Wpass** options under **Design options**.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.

- `dB` — Specify the magnitude in dB (decibels). This is the default setting.

- `Squared` — Specify the magnitude in squared units.

Astop1

Enter the filter attenuation in the first stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow
> Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

> Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).
>
> Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

> Valid when the **Design method** is `equiripple` and you have the DSP System Toolbox installed. Choose one of `Linear`, `Minimum`, or `Maximum`.

Minimum order

> This option only applies when you have the DSP System Toolbox software and **Order mode** is `Minimum`.
>
> Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Wstop1

Weight for the first stopband.

Wpass

Passband weight.

Wstop2

Weight for the second stopband.

Max pole radius

Valid only for IIR designs. Constrains the maximum pole radius. The default is 1. Reducing the max pole radius can produce a transfer function more resistant to quantization.

Init norm

Valid only for IIR designs. The initial norm used in the optimization. The default initial norm is 2.

Init numerator

Specifies an initial estimate of the filter numerator coefficients. This may be useful in difficult optimization problems.

Init denominator

Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Bandstop Filter Design Dialog Box — Main Pane

7-683

filterbuilder

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option so you can enter the filter order.

If you have the DSP System Toolbox software installed, you can specify IIR filters with different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

Frequency constraints
> Select the filter features to use to define the frequency response characteristics. This dialog applies only when **Order mode** is **Specify**.
>
> - `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stop- and passbands.
>
> - `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.
>
> - `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.
>
> - `3dB points` — Define the filter response by specifying the locations of the 3 dB points (IIR filters). The 3 dB point is the frequency for the point 3 dB point below the passband value.
>
> - `3dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband (IIR filters).
>
> - `3dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband (IIR filters).
>
> - `6dB points` — Define the filter response by specifying the locations of the 6-dB points (FIR filters). The 6-dB point is the frequency for the point 6 dB point below the passband value.

Frequency units
> Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

When you design an interpolator, Fs represents the sampling frequency at the filter output rather than the filter input. This option is available only when you set **Filter type** is `interpolator`.

Fpass1

Enter the frequency at the edge of the end of the first passband. Specify the value in either normalized frequency units or the absolute units you select in **Frequency units**.

Fstop1

Enter the frequency at the edge of the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop2

Enter the frequency at the edge of the end of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fpass2

Enter the frequency at the edge of the start of the second passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude constraints

Specify as `Unconstrained` or `Constrained bands`. You must have the DSP System Toolbox software to select `Constrained bands`. Selecting `Constrained bands` enables dialogs for both passbands and the stopband: **Apass1**, **Apass2**, and **Astop**. You cannot specify constraints for all three bands simultaneously.

Setting **Magnitude constraints** to `Constrained bands` enables the **Wstop** and **Wpass** options under **Design options**.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- `Linear` — Specify the magnitude in linear units.

- `dB` — Specify the magnitude in decibels (default).

- `Squared` — Specify the magnitude in squared units.

Apass1

Enter the filter ripple allowed in the first passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels

Apass2

Enter the filter ripple allowed in the second passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the

specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow
Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor
Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint
Valid when the **Design method** is equiripple and you have the DSP System Toolbox installed. Choose one of Linear, Minimum, or Maximum.

Minimum order

This option only applies when you have the DSP System Toolbox software and **Order mode** is `Minimum`.

Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Wpass1

Weight for the first passband.

Wstop

Stopband weight.

Wpass2

Weight for the second passband.

Match exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` .

Max pole radius

Valid only for IIR designs. Constrains the maximum pole radius. The default is 1. Reducing the max pole radius can produce a transfer function more resistant to quantization.

Init norm

Valid only for IIR designs. The initial norm used in the optimization. The default initial norm is 2.

Init numerator

Specifies an initial estimate of the filter numerator coefficients. This may be useful in difficult optimization problems.

Init denominator

Specifies an initial estimate of the filter denominator coefficients. This may be useful in difficult optimization problems.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

CIC Filter Design Dialog Box — Main Pane

CIC Design

CIC Design

Design a Cascaded Integrator-Comb filter.

Save variable as: Hcic   View F

| Main | Data Types | Code Generation |

Filter specifications

Filter type: Decimator ▼   Factor: 2

Differential delay: 1

Frequency specifications

Frequency units: Normalized (0 to 1) ▼

Fpass: .01

Magnitude specifications

Magnitude units: dB ▼

Astop: 60

Filter implementation

☐ Use a System object to implement filter

OK   Cancel   Help

Filter specifications

Parameters in this group enable you to specify your CIC filter format, such as the filter type and the differential delay.

Filter type

> Select whether your filter will be a `decimator` or an `interpolator`. Your choice determines the type of filter and the design methods and structures that are available to implement your filter. Selecting `decimator` or `interpolator` activates the **Factor** option. When you design an interpolator, you enable the **Output Fs** parameter.
>
> When you design either a decimator or interpolator, the resulting filter is a CIC filter that decimates or interpolates your input signal.

Differential Delay

> Specify the differential delay of your CIC filter as an integer value greater than or equal to 1. The default value is 1. The differential delay changes the shape, number, and location of nulls in the filter response. Increasing the differential delay increases the sharpness of the nulls and the response between the nulls. In practice, differential delay values of 1 or 2 are the most common.

Factor

> Specify the decimation or interpolation factor for your filter as an integer value greater than or equal to 1. The default value is 2.

Frequency specifications

Frequency units

> Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Filter implementation

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

CIC Compensator Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the filter order mode and the filter type.

Order mode

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if Specify was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default factor value is 2.

Number of CIC sections

Specify the number of sections in the CIC filter for which you are designing this compensator. Select the number of sections from the drop-down list or enter the number.

Differential Delay

Specify the differential delay of your target CIC filter. The default value is 1. Most CIC filters use 1 or 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve.

Frequency specifications

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Output Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter output. When you provide an output sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available only when you design interpolators.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels

Algorithm

The parameters in this group allow you to specify the design method and structure that filterbuilder uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The

default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

When you select this parameter, the design method determines and design the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select Any, Even, or Odd from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

> **Note** Generally, **Minimum order** designs are not available for IIR filters.

Match exactly

Specifies that the resulting filter design matches either the passband or stopband or both bands when you select `passband` or `stopband` or `both` from the drop-down list.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay.

filterbuilder applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Comb Filter Design Dialog Box—Main Pane

Filter specifications

Parameters in this group enable you to specify the type of comb filter and the number of peaks or notches.

Comb Type

Select Notch or Peak from the drop-down list. Notch creates a comb filter that attenuates a set of harmonically related frequencies. Peak creates a comb filter that amplifies a set of harmonically related frequencies.

Order mode

Select Order or Number of Peaks/Notches from the drop-down menu.

Select Order to enter the desired filter order in the

dialog box. The comb filter has notches or peaks at increments of 2/Order in normalized frequency units.

Select Number of Peaks or Number of Notches to specify the number of peaks or notches and the Shelving filter order

Shelving filter order

The Shelving filter order is a positive integer that determines the sharpness of the peaks or notches. Larger values result in sharper peaks or notches.

Frequency specifications

Parameters in this group enable you to specify the frequency constraints and frequency units.

Frequency specifications

Select Quality factor or Bandwidth.

Quality factor is the ratio of the center frequency of the peak or notch to the bandwidth calculated at the –3 dB point.

Bandwidth specifies the bandwidth of the peak or notch. By default the bandwidth is measured at the –3 dB point. For example, setting the bandwidth equal to 0.1 results in 3 dB frequencies at normalized frequencies 0.05 above and below the center frequency of the peak or notch.

Frequency Units

Specify the frequency units. The default is normalized frequency. Choosing an option in Hz enables the **Input Fs** dialog box.

Magnitude specifications

Specify the units for the magnitude specification and the gain at which the bandwidth is measured. This menu is disabled if you specify a filter order. Select one of the following magnitude units from the drop down list:

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Bandwidth gain — Specify the gain at which the bandwidth is measured. The default is –3 dB.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method
> The IIR Butterworth design is the only option for peaking or notching comb filters.

Filter implementation

Structure
> For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use a System object to implement filter
> Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off.

Differentiator Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order. Graphically, the filter specifications look similar to those shown in the following figure.
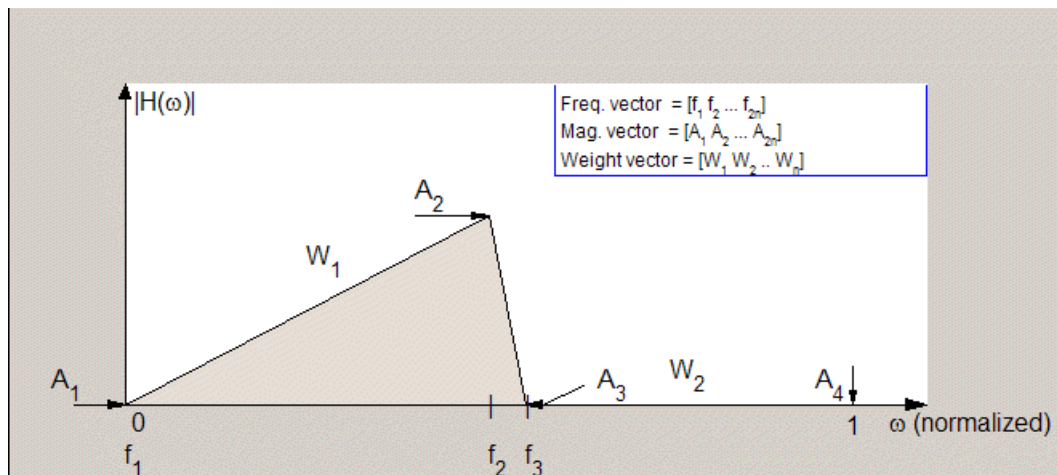
In the figure, regions between specification values such as **Fpass** (f_1) and **Fstop** (f_3) represent transition regions where the filter response is not explicitly defined.

Order mode

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

- Selecting `Decimator` or `Interpolator` activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting `Sample-rate converter` activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve.

Frequency constraints

This option is only available when you specify the order of the filter design. Supported options are `Unconstrained` and `Passband edge` and `stopband edge`.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the

frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude constraints

This option is only available when you specify the order of your filter design. The options for **Magnitude constraints** depend on the value of the **Frequency constraints**. If the value of **Frequency constraints** is Unconstrained, **Magnitude constraints** must be Unconstrained. If the value of **Frequency constraints** is Passband edge and stopband edge, **Magnitude constraints** can be Unconstrained, Passband ripple, or Stopband attenuation.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop2

Enter the filter attenuation in the second stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that filterbuilder uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Wpass

Passband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to `Passband edge and stopband edge` and the **Design method** is `Equiripple`.

Wstop

Stopband weight. This option is only available for a specified-order design when **Frequency constraints** is equal to `Passband edge and stopband edge` and the **Design method** is `Equiripple`.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned

off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Fractional Delay Filter Design Dialog Box — Main Pane
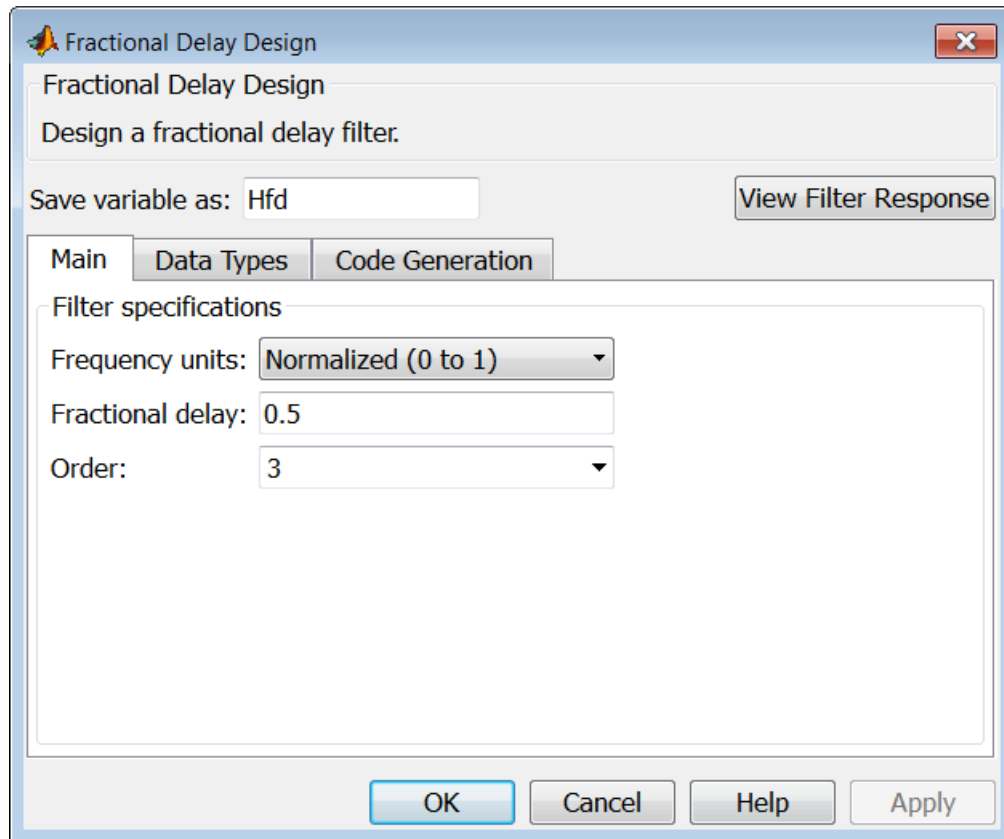
Frequency specifications

Parameters in this group enable you to specify your filter format, such as the fractional delay and the filter order.

Order

If you choose Specify for **Order mode**, enter your filter
order in this field, or select the order from the drop-down
list.filterbuilder designs a filter with the order you specify.

Fractional delay

Specify a value between 0 and 1 samples for the filter fractional
delay. The default value is 0.5 samples.

Frequency units

Use this parameter to specify whether your frequency settings are
normalized or in absolute frequency. Select Normalized (0 1)
to enter frequencies in normalized form. This behavior is the
default. To enter frequencies in absolute values, select one of the
frequency units from the drop-down list—Hz, kHz, MHz, or GHz.
Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**,
defines the sampling frequency at the filter input. When you
provide an input sampling frequency, all frequencies in the
specifications are in the selected units as well. This parameter is
available when you select one of the frequency options from the
Frequency units list.

Halfband Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter type and order.

Impulse response

> Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

> **Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

> Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

> Select Single-rate, Decimator, or Interpolator. By default, filterbuilder specifies single-rate filters.

> When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that decimates or interpolates your input by a factor of two.

Order

> Enter the filter order. This option is enabled only if Specify was selected for **Order mode**.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications for a halfband lowpass filter look similar to those shown in the following figure.

In the figure, the transition region lies between the end of the passband and the start of the stopband. The width is defined explicitly by the value of **Transition width**.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width
> Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units
> Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.
>
> - Linear — Specify the magnitude in linear units.
> - dB — Specify the magnitude in decibels (default).

Astop
> Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that filterbuilder uses to implement your filter.

Design Method
> Lists the design methods available for the frequency and magnitude specifications you entered. For FIR halfband filters, the available design options are Equiripple and Kaiser window. For IIR halfband filters, the available design options are Butterworth, Elliptic, and IIR quasi-linear phase.

Design Options

The following design options are available for FIR halfband filters when the user specifies an equiripple design:

Minimum phase

To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- Flat — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- Linear — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- 1/f — Specifies that the stopband attenuation changes exponentially as the frequency increases, where f is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to Flat, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to Linear, enter the slope of the stopband in units of dB/rad/s. filterbuilder applies that slope to the stopband.

- When you set **Stopband shape** to 1/f, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. filterbuilder applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter implementation

Structure
> For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter.

Use a System object to implement filter
> Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Highpass Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option so you can enter the filter order.

If your **Impulse response** is IIR, you can specify an equal order for the numerator and denominator, or different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type

This option is only available if you have the DSP System Toolbox software. Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a highpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the interpolation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

In the figure, the region between specification values Fstop and Fpass represents the transition region where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Stopband edge and passband edge` — Define the filter by specifying the frequencies for the edges for the stopband and passband.

- `Passband edge` — Define the filter by specifying the frequency for the edge of the passband.

- `Stopband edge` — Define the filter by specifying the frequency for the edges of the stopband.

- `Stopband edge and 3dB point` — Define the filter by specifying the stopband edge frequency and the 3-dB down point (IIR designs).

- `3dB point and passband edge` — Define the filter by specifying the 3-dB down point and passband edge frequency (IIR designs).

- `3dB point` — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).

- `6dB point` — Define the filter by specifying the frequency for the 6-dB point in the filter response (FIR designs).

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the

specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default).

- Squared — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

>   Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

>   Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

>   Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).
>
>   Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a

reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

This option only applies when you have the DSP System Toolbox software and when the **Design method** is `equiripple`. Select one of `Linear`, `Minimum`, or `Maximum`.

Minimum order — This option only applies when you have the DSP System Toolbox software and the **Order mode** is `Minimum`.

Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Match Exactly

Specifies that the resulting filter design matches either the passband or stopband when you select `Passband` or `Stopband`.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Wpass
> Passband weight. This option only applies when **Impulse response** is FIR and **Order mode** is `Specify`.

Wstop
> Stopband weight. This option only applies when **Impulse response** is FIR and **Order mode** is `Specify`.

Filter implementation

Structure
> For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter
> Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Hilbert Filter Design Dialog Box — Main Pane

7-729

Impulse response
> Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

> ---
> **Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.
> ---

Order mode
> This option is only available if you have the DSP System Toolbox software. Select either Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type
> This option is only available if you have the DSP System Toolbox software. Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

> • Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

> • Selecting Sample-rate converter activates both factors.

> When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order
> Enter the filter order. This option is enabled only if Specify was selected for **Order mode**.
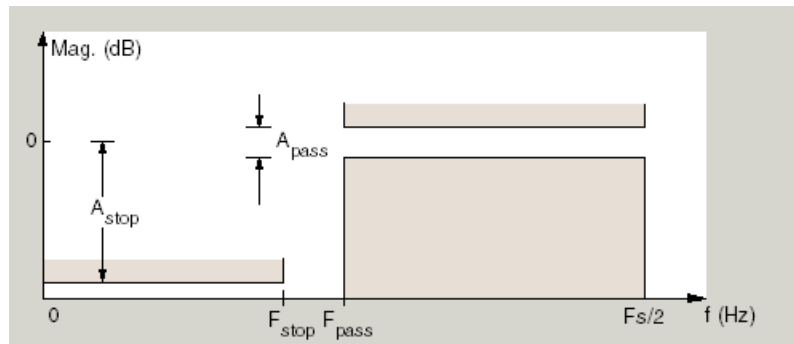
Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

In the figure, the regions between 0 and f_1 and between f_2 and 1 represent the transition regions where the filter response is explicitly defined by the transition width.

filterbuilder

Frequency units

> Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

> Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

> Specify the width of the transitions at the ends of the passband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

> Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.
>
> - `Linear` — Specify the magnitude in linear units.
>
> - `dB` — Specify the magnitude in decibels (default)
>
> - `Squared` — Specify the magnitude in squared units.

Apass

> Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method

> Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

> Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

> Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).
>
> Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a

reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

FIR Type

This option is only available in a minimum-order design. Specify whether to design a type 3 or a type 4 FIR filter. The filter type is defined as follows:

- Type 3 — FIR filter with even order antisymmetric coefficients

- Type 4 — FIR filter with odd order antisymmetric coefficients
Select 3 or 4 from the drop-down list.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Inverse Sinc Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Order mode

    Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Response type

    Select Lowpass or Highpass to design an inverse sinc lowpass or highpass filter.

Filter type

    Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

    When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

    Enter the filter order. This option is enabled only if Specify was selected for **Order mode**.

Decimation Factor

    Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

Interpolation Factor

    Enter the decimation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

Regions between specification values such as Fpass and Fstop represent transition regions where the filter response is not explicitly defined.

Frequency constraints

This option is only available when you specify the filter order. The following options are available:

- Passband and stopband edges — Define the filter by specifying the frequencies for the edges for the stop- and passbands.

- Passband edge — Define the filter by specifying frequencies for the edges of the passband.

- Stopband edge — Define the filter by specifying frequencies for the edges of the stopbands.

- 6dB point — The 6-dB point is the frequency for the point 6 dB point below the passband value.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select Normalized (0 1) to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—Hz, kHz, MHz, or GHz. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the end of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default)

- Squared — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that filterbuilder uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint

Available options are `Linear`, `Minimum`, and `Maximum`.

Stopband Shape

Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options;

- `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

- `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

- `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Sinc frequency factor

A frequency dilation factor. The sinc frequency factor, C, parameterizes the passband magnitude response for a lowpass design through $H(\omega) = \text{sinc}(C\omega)^{(-P)}$ and for a highpass design through $H(\omega) = \text{sinc}(C(1-\omega))^{(-P)}$.

Sinc power

Negative power of passband magnitude response. The sinc power, P, parameterizes the passband magnitude response for a lowpass design through $H(\omega) = \text{sinc}(C\omega)^{(-P)}$ and for a highpass design through $H(\omega) = \text{sinc}(C(1-\omega))^{(-P)}$.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Lowpass Filter Design Dialog Box — Main Pane

Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Impulse response

Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.

Note The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.

Order mode

Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

If your **Impulse response** is IIR, you can specify an equal order for the numerator and denominator, or different numerator and denominator orders. The default is equal orders. To specify a different denominator order, check the **Denominator order** box.

Filter type

This option is only available if you have the DSP System Toolbox. Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.

- Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.

- Selecting Sample-rate converter activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if Specify was selected for **Order mode**.

Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Decimator or Sample-rate converter. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to Interpolator or Sample-rate converter. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to the one shown in the following figure.

In the figure, regions between specification values such as F_{pass} and F_{stop} represent transition regions where the filter response is not explicitly defined.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edge` — Define the filter by specifying the frequencies for the edge of the stopband and passband.

- `Passband edge` — Define the filter by specifying the frequency for the edge of the passband.

- `Stopband edge` — Define the filter by specifying the frequency for the edges of the stopband.

- `Passband edge and 3dB point` — Define the filter by specifying the passband edge frequency and the 3-dB down point (IIR designs).

- `3dB point and stopband edge` — Define the filter by specifying the 3-dB down point and stopband edge frequency (IIR designs).

- `3dB point` — Define the filter by specifying the frequency for the 3-dB point (IIR designs or maxflat FIR).

- `6dB point` — Define the filter by specifying the frequency for the 6-dB point in the filter response (FIR designs).

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is

available when you select one of the frequency options from the **Frequency units** list.

Fpass

Enter the frequency at the of the passband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Fstop

Enter the frequency at the start of the stopband. Specify the value in either normalized frequency units or the absolute units you select **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.

- dB — Specify the magnitude in decibels (default)

- Squared — Specify the magnitude in squared units.

Apass

Enter the filter ripple allowed in the passband in the units you choose for **Magnitude units**, either linear or decibels.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that filterbuilder uses to implement your filter.

Design Method

> Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

> Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

> Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).
>
> Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Phase constraint
> This option only applies when you have the DSP System Toolbox software and when the **Design method** is `equiripple`. Select one of `Linear`, `Minimum`, or `Maximum`.

Minimum order — This option only applies when you have the DSP System Toolbox software and the **Order mode** is `Minimum`.
> Select `Any` (default), `Even`, or `Odd`. Selecting `Even` or `Odd` forces the minimum-order design to be an even or odd order.

Match Exactly
> Specifies that the resulting filter design matches either the passband or stopband when you select `Passband` or `Stopband`.

Stopband Shape
> Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:
>
> - `Flat` — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.
>
> - `Linear` — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.
>
> - `1/f` — Specifies that the stopband attenuation changes exponentially as the frequency increases, where `f` is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay
> When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:
>
> - When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.
>
> - When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Wpass

Passband weight. This option only applies when **Impulse response** is `FIR` and **Order mode** is `Specify`.

Wstop

Stopband weight. This option only applies when **Impulse response** is `FIR` and **Order mode** is `Specify`.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Notch

See "Peak/Notch Filter Design Dialog Box — Main Pane" on page 7-767.

Nyquist Filter Design Dialog Box — Main Pane
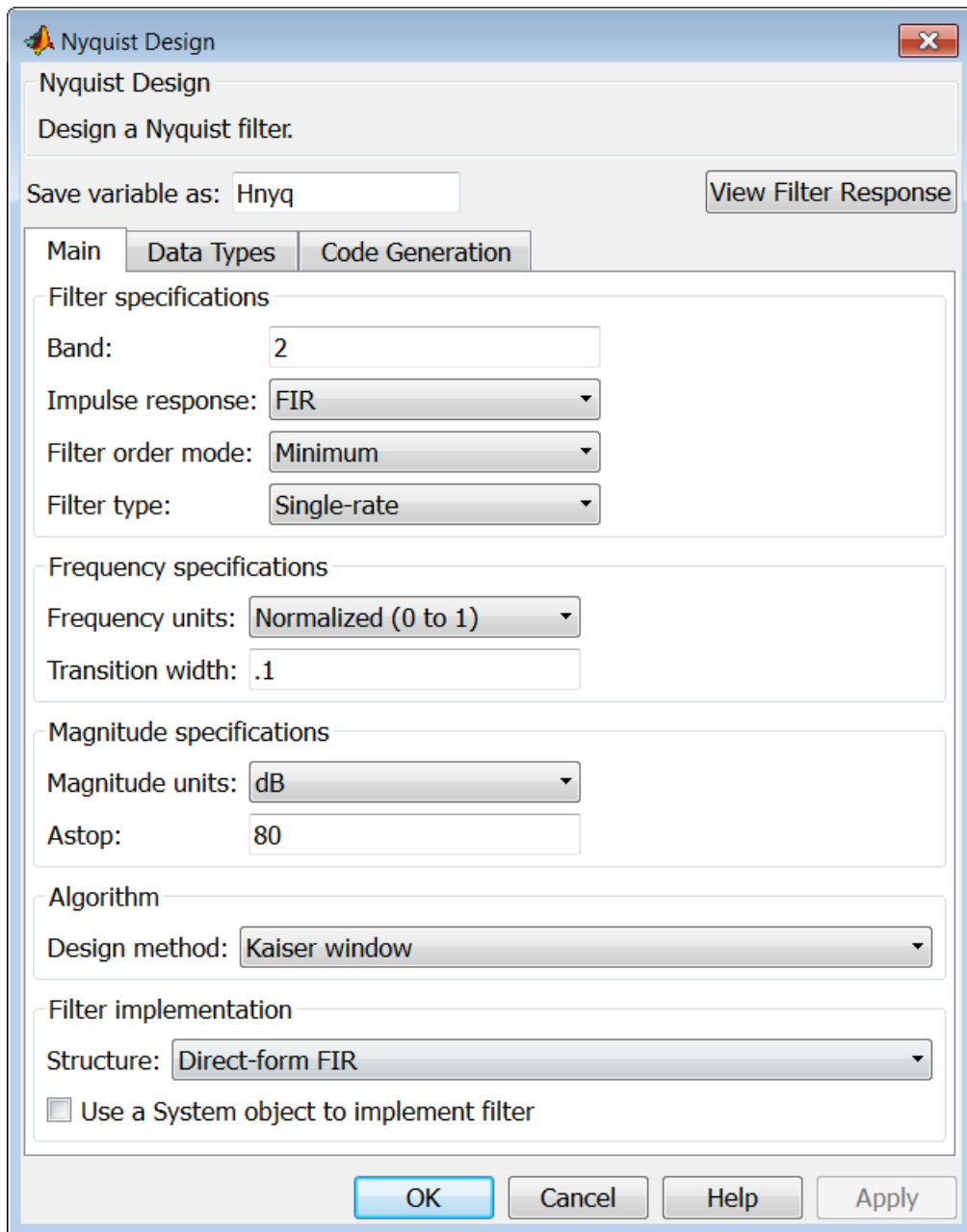
Filter specifications

Parameters in this group enable you to specify your filter format, such as the impulse response and the filter order.

Band

> Specifies the location of the center of the transition region between the passband and the stopband. The center of the transition region, bw, is calculated using the value for Band:
>
> bw = Fs/(2*Band).

Impulse response

> Select FIR or IIR from the drop-down list, where FIR is the default impulse response. When you choose an impulse response, the design methods and structures you can use to implement your filter change accordingly.
>
> ---
>
> **Note** The design methods and structures for FIR filters are not the same as the methods and structures for IIR filters.
>
> ---

Order mode

> Select Minimum (the default) or Specify from the drop-down list. Selecting Specify enables the **Order** option (see the following sections) so you can enter the filter order.

Filter type

> Select Single-rate, Decimator, Interpolator, or Sample-rate converter. Your choice determines the type of filter as well as the design methods and structures that are available to implement your filter. By default, filterbuilder specifies single-rate filters.
>
> - Selecting Decimator or Interpolator activates the **Decimation Factor** or the **Interpolation Factor** options respectively.
>
> - Selecting Sample-rate converter activates both factors.

When you design either a decimator or an interpolator, the resulting filter is a bandpass filter that either decimates or interpolates your input signal.

Order

Enter the filter order. This option is enabled only if `Specify` was selected for **Order mode**.
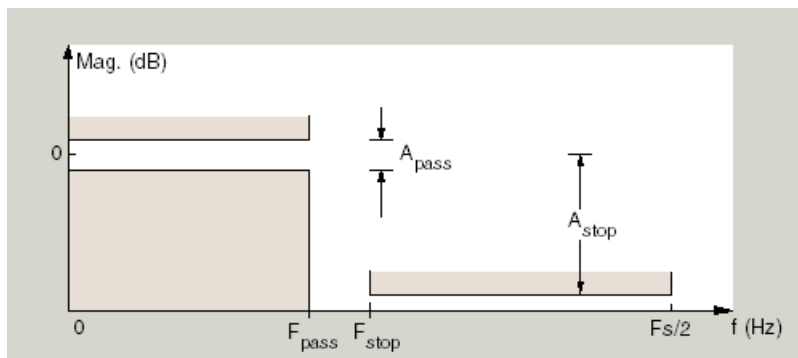
Decimation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Decimator` or `Sample-rate converter`. The default factor value is 2.

Interpolation Factor

Enter the decimation factor. This option is enabled only if the **Filter type** is set to `Interpolator` or `Sample-rate converter`. The default factor value is 2.

Frequency specifications

The parameters in this group allow you to specify your filter response curve. Graphically, the filter specifications look similar to those shown in the following figure.

In the figure, BW is the width of the transition region and **Band** determines the location of the center of the region.

Frequency constraints

Select the filter features to use to define the frequency response characteristics. The list contains the following options, when available for the filter specifications.

- `Passband and stopband edges` — Define the filter by specifying the frequencies for the edges for the stopbands and passbands.

- `Passband edges` — Define the filter by specifying frequencies for the edges of the passband.

- `Stopband edges` — Define the filter by specifying frequencies for the edges of the stopbands.

- `3 dB points` — Define the filter response by specifying the locations of the 3 dB points. The 3 dB point is the frequency for the point 3 dB point below the passband value.

- `3 dB points and passband width` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the passband.

- `3 dB points and stopband widths` — Define the filter by specifying frequencies for the 3 dB points in the filter response and the width of the stopband.

Frequency units

Use this parameter to specify whether your frequency settings are normalized or in absolute frequency. Select `Normalized (0 1)` to enter frequencies in normalized form. This behavior is the default. To enter frequencies in absolute values, select one of the frequency units from the drop-down list—`Hz`, `kHz`, `MHz`, or `GHz`. Selecting one of the unit options enables the **Input Fs** parameter.

Input Fs

Fs, specified in the units you selected for **Frequency units**, defines the sampling frequency at the filter input. When you provide an input sampling frequency, all frequencies in the specifications are in the selected units as well. This parameter is available when you select one of the frequency options from the **Frequency units** list.

Transition width

Specify the width of the transition between the end of the passband and the edge of the stopband. Specify the value in normalized frequency units or the absolute units you select in **Frequency units**.

Magnitude specifications

The parameters in this group let you specify the filter response in the passbands and stopbands.

Magnitude units

Specify the units for any parameter you provide in magnitude specifications. Select one of the following options from the drop-down list.

- Linear — Specify the magnitude in linear units.
- dB — Specify the magnitude in decibels (default)
- Squared — Specify the magnitude in squared units.

Astop

Enter the filter attenuation in the stopband in the units you choose for **Magnitude units**, either linear or decibels.

Algorithm

The parameters in this group allow you to specify the design method and structure that filterbuilder uses to implement your filter.

Design Method

Lists the design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter, such as changing the impulse response, the methods available to design filters changes as well. The default IIR design method is usually Butterworth, and the default FIR method is equiripple.

Scale SOS filter coefficients to reduce chance of overflow

Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Design Options

The options for each design are specific for each design method. This section does not present all of the available options for all designs and design methods. There are many more that you encounter as you select different design methods and filter specifications. The following options represent some of the most common ones available.

Density factor

Density factor controls the density of the frequency grid over which the design method optimization evaluates your filter response function. The number of equally spaced points in the grid is the value you enter for **Density factor** times (filter order + 1).

Increasing the value creates a filter that more closely approximates an ideal equiripple filter but increases the time required to design the filter. The default value of 16 represents a reasonable trade between the accurate approximation to the ideal filter and the time to design the filter.

Minimum phase

> To design a filter that is minimum phase, select **Minimum phase**. Clearing the **Minimum phase** option removes the phase constraint—the resulting design is not minimum phase.

Minimum order

> When you select this parameter, the design method determines and designs the minimum order filter to meet your specifications. Some filters do not provide this parameter. Select Any, Even, or Odd from the drop-down list to direct the design to be any minimum order, or minimum even order, or minimum odd order.

> ---
> **Note** Generally, **Minimum order** designs are not available for IIR filters.
> ---

Match Exactly

> Specifies that the resulting filter design matches either the passband or stopband or both bands when you select passband or stopband or both from the drop-down list.

Stopband Shape

> Stopband shape lets you specify how the stopband changes with increasing frequency. Choose one of the following options:

> - Flat — Specifies that the stopband is flat. The attenuation does not change as the frequency increases.

> - Linear — Specifies that the stopband attenuation changes linearly as the frequency increases. Change the slope of the stopband by setting **Stopband decay**.

> - 1/f — Specifies that the stopband attenuation changes exponentially as the frequency increases, where f is the frequency. Set the power (exponent) for the decay in **Stopband decay**.

Stopband Decay

When you set Stopband shape, Stopband decay specifies the amount of decay applied to the stopband. the following conditions apply to Stopband decay based on the value of Stopband Shape:

- When you set **Stopband shape** to `Flat`, **Stopband decay** has no affect on the stopband.

- When you set **Stopband shape** to `Linear`, enter the slope of the stopband in units of dB/rad/s. `filterbuilder` applies that slope to the stopband.

- When you set **Stopband shape** to `1/f`, enter a value for the exponent n in the relation $(1/f)^n$ to define the stopband decay. `filterbuilder` applies the $(1/f)^n$ relation to the stopband to result in an exponentially decreasing stopband attenuation.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure, and IIR filters use direct-form II filters with SOS.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Octave Filter Design Dialog Box — Main Pane

Filter specifications

Order

Specify filter order. Possible values are: `4, 6, 8, 10`.

Bands per octave

Specify the number of bands per octave. Possible values are: `1, 3, 6, 12, 24`.

Frequency units

Specify frequency units as `Hz` or `kHz`.

Input Fs

Specify the input sampling frequency in the frequency units specified previously.

Center Frequency

Select from the drop-down list of available center frequency values.

Algorithm

Design Method

Butterworth is the design method used for this type of filter.

Scale SOS filter coefficients to reduce chance of overflow

Select the check box to scale the filter coefficients.

Filter implementation

Structure

Specify filter structure. Choose from:

- Direct-form I SOS

- Direct-form II SOS

- Direct-form I transposed SOS

- Direct-form II transposed SOS

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default, the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Parametric Equalizer Filter Design Dialog Box — Main Pane

filterbuilder

Filter specifications

Order mode

Select Minimum to design a minimum order filter that meets the design specifications, or Specify to enter a specific filter order. The order mode also affects the possible frequency constraints, which in turn limit the gain specifications. For example, if you specify a Minimum order filter, the available frequency constraints are:

- Center frequency, bandwidth, passband width

- Center frequency, bandwidth, stopband width

If you select Specify, the available frequency constraints are:

- Center frequency, bandwidth

- Center frequency, quality factor

- Shelf type, cutoff frequency, quality factor

- Shelf type, cutoff frequency, shelf slope parameter

- Low frequency, high frequency

Order

This parameter is enabled only if the **Order mode** is set to Specify. Enter the filter order in this text box.

Frequency specifications

Depending on the filter order, the possible frequency constraints change. Once you choose the frequency constraints, the input boxes in this area change to reflect the selection.

Frequency constraints

Select the specification to represent the frequency constraints. The following options are available:

- Center frequency, bandwidth, passband width (available for minimum order only)

- `Center frequency, bandwidth, stopband width` (available for minimum order only)

- `Center frequency, bandwidth` (available for a specified order only)

- `Center frequency, quality factor` (available for a specified order only)

- `Shelf type, cutoff frequency, quality factor` (available for a specified order only)

- `Shelf type, cutoff frequency, shelf slope parameter` (available for a specified order only)

- `Low frequency, high frequency` (available for a specified order only)

Frequency units

Select the frequency units from the available drop down list (`Normalized, Hz, kHz, MHz, GHz`). If `Normalized` is selected, then the **Input Fs** box is disabled for input.

Input Fs

Enter the input sampling frequency. This input box is disabled for input if `Normalized` is selected in the **Frequency units** input box.

Center frequency

Enter the center frequency in the units specified by the value in **Frequency units**.

Bandwidth

The bandwidth determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Bandwidth gain** in the **Gain specifications** section. By default, the **Bandwidth gain** defaults to `db(sqrt(.5))`, or −3 dB relative to the center frequency. The **Bandwidth** property only applies when the **Frequency constraints** are: `Center frequency, bandwidth, passband width`, `Center frequency, bandwidth, stopband width`, or `Center frequency, bandwidth`.

Passband width

> The passband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Passband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, passband width`.

Stopband width

> The stopband width determines the frequency points at which the filter magnitude is attenuated by the value specified as the **Stopband gain** in the **Gain specifications** section. This option is enabled only if the filter is of minimum order, and the frequency constraint selected is `Center frequency, bandwidth, stopband width`.

Low frequency

> Enter the low frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

High frequency

> Enter the high frequency cutoff. This option is enabled only if the filter order is user specified and the frequency constraint selected is `Low frequency, high frequency`. The filter magnitude is attenuated by the amount specified in **Bandwidth gain**.

Gain specifications

Depending on the filter order and frequency constraints, the possible gain constraints change. Also, once you choose the gain constraints the input boxes in this area change to reflect the selection.

Gain constraints

> Select the specification array to represent gain constraints, and remember that not all of these options are available for all configurations. The following is a list of all available options:

- Reference, center frequency, bandwidth, passband

- Reference, center frequency, bandwidth, stopband

- Reference, center frequency, bandwidth,passband, stopband

- Reference, center frequency, bandwidth

Gain units

Specify the gain units either dB or squared. These units are used for all gain specifications in the dialog box.

Reference gain

The reference gain determines the level to which the filter magnitude attenuates in **Gain units**. The reference gain is a *floor* gain for the filter magnitude response. For example, you may use the reference gain together with the **Center frequency gain** to leave certain frequencies unattenuated (reference gain of 0 dB) while boosting other frequencies.

Bandwidth gain

Specifies the gain in **Gain units** at which the bandwidth is defined. This property applies only when the **Frequency constraints** specification contains a bandwidth parameter, or is Low frequency, high frequency.

Center frequency gain

Specify the center frequency in **Gain units**

Passband gain

The passband gain determines the level in **Gain units** at which the passband is defined. The passband is determined either by the **Passband width** value, or the **Low frequency** and **High frequency** values in the **Frequency specifications** section.

Stopband gain

The stopband gain is the level in **Gain units** at which the stopband is defined. This property applies only when the **Order mode** is minimum and the **Frequency constaints** are Center frequency, bandwidth, stopband width.

Boost/cut gain

> The boost/cut gain applies only when the designing a shelving filter. Shelving filters include the `Shelf type` parameter in the **Frequency constraints** specification. The gain in the passband of the shelving filter is increased by **Boost/cut gain** dB from a *floor* gain of 0 dB.

Algorithm

Design method

> Select the design method from the drop-down list. Different IIR design methods are available depending on the filter constraints you specify.

Scale SOS filter coefficients to reduce chance of overflow

> Select the check box to scale the filter coefficients.

Filter implementation

Structure

> Select filter structure. The possible choices are:
>
> - Direct-form I SOS
>
> - Direct-form II SOS
>
> - Direct-form I transposed SOS
>
> - Direct-form II transposed SOS

Use a System object to implement filter

> Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Peak/Notch Filter Design Dialog Box — Main Pane

filterbuilder

Filter specifications

In this area you can specify whether you want to design a peaking filter or a notching filter, as well as the order of the filter.

Response
> Select Peak or Notch from the drop-down box.

Order
> Enter the filter order. The order must be even.

Frequency specifications

This group of parameters allows you to specify frequency constraints and units.

Frequency Constraints
> Select the frequency constraints for filter specification. There are two choices as follows:
>
> - Center frequency and quality factor
> - Center frequency and bandwidth

Frequency units
> The frequency units are normalized by default. If you specify units other than normalized, filterbuilder assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: Normalized (0 to 1), Hz, kHz, MHz, GHz.

Input Fs
> This input box is enabled if **Frequency units** other than Normalized (0 to 1) are specified. Enter the input sampling frequency.

Center frequency
> Enter the center frequency in the units you specified in **Frequency units**.

Quality Factor

This input box is enabled only when Center frequency and quality factor is chosen for the **Frequency Constraints**. Enter the quality factor.

Bandwidth

This input box is enabled only when Center frequency and bandwidth is chosen for the **Frequency Constraints**. Enter the bandwidth.

Magnitude specifications

This group of parameters allows you to specify the magnitude constraints, as well as their values and units.

Magnitude Constraints

Depending on the choice of constraints, the other input boxes are enabled or disabled. Select from four magnitude constraints available:

- Unconstrained

- Passband ripple

- Stopband attenuation

- Passband ripple and stopband attenuation

Magnitude units

Select the magnitude units: either dB or squared.

Apass

This input box is enabled if the magnitude constraints selected are Passband ripple or Passband ripple and stopband attenuation. Enter the passband ripple.

Astop

This input box is enabled if the magnitude constraints selected are Stopband attenuation or Passband ripple and stopband attenuation. Enter the stopband attenuation.

Algorithm

The parameters in this group allow you to specify the design method and structure that `filterbuilder` uses to implement your filter.

Design Method
> Lists all design methods available for the frequency and magnitude specifications you entered. When you change the specifications for a filter the methods available to design filters changes as well.

Scale SOS filter coefficients to reduce chance of overflow
> Selecting this parameter directs the design to scale the filter coefficients to reduce the chances that the inputs or calculations in the filter overflow and exceed the representable range of the filter. Clearing this option removes the scaling. This parameter applies only to IIR filters.

Filter implementation

Structure
> Lists all available filter structures for the filter specifications and design method you select. The typical options are:
>
> - Direct-form I SOS
>
> - Direct-form II SOS
>
> - Direct-form I transposed SOS
>
> - Direct-form II transposed SOS

Use a System object to implement filter
> Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Pulse-shaping Filter Design Dialog Box—Main Pane

Filter specifications

Parameters in this group enable you to specify the shape and length of the filter.

Pulse shape

Select the shape of the impulse response from the following options:

- `Raised Cosine`

- `Square Root Raised Cosine`

- `Gaussian`

Order mode

This specification is only available for raised cosine and square root raised cosine filters. For these filters, select one of the following options:

- `Minimum`— This option will result in the minimum-length filter satisfying the user-specified **Frequency specifications**.

- `Specify order`—This option allows the user to construct a raised cosine or square root cosine filter of a specified order by entering an even number in the **Order** input box. The length of the impulse response will be `Order+1` .

- `Specify symbols`—This option enables the user to specify the length of the impulse response in an alternative manner. If `Specify symbols` is chosen, the **Order** input box changes to the **Number of symbols** input box.

Samples per symbol

Specify the oversampling factor. Increasing the oversampling factor guards against aliasing and improves the FIR filter approximation to the ideal frequency response. If **Order** is specified in **Number of symbols**, the filter length will be **Number of symbols*Samples per symbol**+1. The product **Number of symbols*Samples per symbol** must be an even number.

If a Gaussian filter is specified, the filter length must be specified in **Number of symbols** and **Samples per symbol**. The product **Number of symbols*Samples per symbol** must be an even number. The filter length will be **Number of symbols*Samples per symbol**+1.

Filter Type

This option is only available if you have the DSP System Toolbox software. Choose `Single rate`, `Decimator`, `Interpolator`, or `Sample-rate converter`. If you select `Decimator` or `Interpolator`, the decimation and interpolation factors default to the value of the **Samples per symbol**. If you select `Sample-rate converter`, the interpolation factor defaults to **Samples per symbol** and the decimation factor defaults to 3.

Frequency specifications

Parameters in this group enable you to specify the frequency response of the filter. For raised cosine and square root raised cosine filters, the frequency specifications include:

Rolloff factor

The rolloff factor takes values in the range [0,1]. The smaller the rolloff factor, the steeper the transition in the stopband.

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterbuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1)`, `Hz`, `kHz`, `MHz`, `GHz`

For a Gaussian pulse shape, the available frequency specifications are:

Bandwidth-time product

This option allows the user to specify the width of the Gaussian filter. Note that this is independent of the length of the filter. The bandwidth-time product (BT) must be a positive real number.

Smaller values of the bandwidth-time product result in larger pulse widths in time and steeper stopband transitions in the frequency response.

Frequency units

The frequency units are normalized by default. If you specify units other than normalized, `filterbuilder` assumes that you wish to specify an input sampling frequency, and enables this input box. The choice of frequency units are: `Normalized (0 to 1)`, `Hz`, `kHz`, `MHz`, `GHz`

Magnitude specifications

If the **Order mode** is specified as `Minimum`, the **Magnitude units** may be selected from:

- `dB`—Specify the magnitude in decibels (default).

- `Linear`—Specify the magnitude in linear units.

Algorithm

The only **Design method** available for FIR pulse-shaping filters is the `Window` method.

Filter implementation

Structure

For the filter specifications and design method you select, this parameter lists the filter structures available to implement your filter. By default, FIR filters use direct-form structure.

Use a System object to implement filter

Selecting this check box gives you the choice of using a system object to implement the filter. By default the check box is turned off. When the current design method or structure is not supported by a system object filter, then this check box is disabled.

Purpose   Store CIC filter states

Description   `filtstates.cic` objects hold the states information for CIC filters. Once you create a CIC filter, the states for the filter are stored in `filtstates.cic` objects, and you can access them and change them as you would any property of the filter. This arrangement parallels that of the `filtstates` object that IIR direct-form I filters use (refer to `filtstates` for more information).

Each `States` property in the CIC filter comprises two properties — `Numerator` and `Comb` — that hold `filtstates.cic` objects. Within the `filtstates.cic` objects are the numerator-related and comb-related filter states. The states are column vectors, where each column represents the states for one section of the filter. For example, a CIC filter with four decimator sections and four interpolator sections has `filtstates.cic` objects that contain four columns of states each.

Examples   To show you the `filtstates.cic` object, create a CIC decimator and filter a signal:

```
fs = 44.1e3;             % Original sampling frequency: 44.1kHz.
n = 0:10239;             % 10240 samples, 0.232 second long signal.
x  = sin(2*pi*1e3/fs*n); % Original signal, sinusoid at 1kHz.
hm=mfilt.cicdecim(5,2,4);
y=filter(hm,x);
```

`hm` has nonzero states now.

You can use `int` to see the states as 32-bit integers.

```
s=hm.states;
int(s.Integrator)
```

See Also   `mfilt` | `mfilt.cicdecim` | `mfilt.cicinterp` | `filtstates`

fircband

Purpose　　　　Constrained-band equiripple FIR filter

Syntax　　　　```
b = fircband(n,f,a,w,c)
b = fircband(n,f,a,s)
b = fircband(...,'1')
b = fircband(...,'minphase')
b = fircband(..., 'check')
b = fircband(...,{lgrid})
[b,err] = fircband(...)
[b,err,res] = fircband(...)
```

**Description**　　`fircband` is a minimax filter design algorithm that you use to design the following types of real FIR filters:

- Types 1-4 linear phase

  - Type 1 is even order, symmetric

  - Type 2 is odd order, symmetric

  - Type 3 is even order, antisymmetric

  - Type 4 is odd order, antisymmetric

- Minimum phase

- Maximum phase

- Minimum order (even or odd), extra ripple

- Maximal ripple

- Constrained ripple

- Single-point band (notching and peaking)

- Forced gain

- Arbitrary shape frequency response curve filters

`b = fircband(n,f,a,w,c)` designs filters having constrained error magnitudes (ripples). `c` is a cell array of strings of the same length as `w`. The entries of `c` must be either 'c' to indicate that the corresponding element in `w` is a constraint (the ripple for that band cannot exceed that

value) or 'w' indicating that the corresponding entry in w is a weight. There must be at least one unconstrained band — c must contain at least one w entry. For instance, Example 1 below uses a weight of one in the passband, and constrains the stopband ripple not to exceed 0.2 (about 14 dB).

A hint about using constrained values: if your constrained filter does not touch the constraints, increase the error weighting you apply to the unconstrained bands.

Notice that, when you work with constrained stopbands, you enter the stopband constraint according to the standard conversion formula for power — the resulting filter attenuation or constraint equals 20*log(*constraint*) where *constraint* is the value you enter in the function. For example, to set 20 dB of attenuation, use a value for the constraint equal to 0.1. This applies to constrained stopbands only.

b = fircband(n,f,a,s) is used to design filters with special properties at certain frequency points. s is a cell array of strings and must be the same length as f and a. Entries of s must be one of:

- 'n' — normal frequency point.

- 's' — single-point band. The frequency band is given by a single point. You must specify the corresponding gain at this frequency point in a.

- 'f' — forced frequency point. Forces the gain at the specified frequency band to be the value specified.

- 'i' — indeterminate frequency point. Use this argument when bands abut one another (no transition region).

b = fircband(...,'1') designs a type 1 filter (even-order symmetric). You could also specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), or type 4 (odd-order antisymmetric) filters. Note there are restrictions on a at f = 0 or f = 1 for types 2, 3, and 4.

b = fircband(...,'minphase') designs a minimum-phase FIR filter. There is also 'maxphase'.

b = fircband(..., '**check**') produces a warning when there are potential transition-region anomalies in the filter response.

b = fircband(...,{lgrid}), where {lgrid} is a scalar cell array containing an integer, controls the density of the frequency grid.

[b,err] = fircband(...) returns the unweighted approximation error magnitudes. err has one element for each independent approximation error.

[b,err,res] = fircband(...) returns a structure res of optional results computed by fircband, and contains the following fields:

| Structure Field | Contents |
| --- | --- |
| res.fgrid | Vector containing the frequency grid used in the filter design optimization |
| res.des | Desired response on fgrid |
| res.wt | Weights on fgrid |
| res.h | Actual frequency response on the frequency grid |
| res.error | Error at each point (desired response - actual response) on the frequency grid |
| res.iextr | Vector of indices into fgrid of external frequencies |
| res.fextr | Vector of extremely frequencies |
| res.order | Filter order |
| res.edgecheck | Transition-region anomaly check. One element per band edge. Element values have the following meanings: 1 = OK , 0 = probable transition-region anomaly , -1 = edge not checked. Computed when you specify the 'check' input option in the function syntax. |

| Structure Field | Contents |
|---|---|
| res.iterations | Number of Remez iterations for the optimization |
| res.evals | Number of function evaluations for the optimization |

**Examples**  Two examples of designing filters with constrained bands.

### Example 1

design a 12th-order lowpass filter with a constraint on the filter response.

```
b = fircband(12,[0 0.4 0.5 1], [1 1 0 0], ...
[1 0.2], {'w' 'c'});
```

Using fvtool to display the result b shows you the response of the filter you designed.

Magnitude Response (dB)

Example 2

design two filters of different order with the stopband constrained to 60 dB. Use excess order (80) in the second filter to improve the passband ripple.

```
b1=fircband(60,[0 .2 .25 1],[1 1 0 0],...
[1 .001],{'w','c'});
b2=fircband(80,[0 .2 .25 1],[1 1 0 0],...
[1 .001],{'w','c'});
hfvt = fvtool(b1,1,b2,1);
legend(hfvt,'Filter Order 60','Filter Order 80');
```

To set the stopband constraint to 60 dB, enter 0.001, since $20*\log(0.001) = -60$, or 60 dB of signal attenuation.

See Also   firceqrip | firgr | firls | firpm

fireqint

Purpose
Equiripple FIR interpolators

Syntax
```
b = fireqint(n,l,alpha)
b = fireqint(n,l,alpha,w)
b = fireqint('minorder',l,alpha,r)
b = fireqint({'minorder',initord},l,alpha,r)
```

Description
`b = fireqint(n,l,alpha)` designs an FIR equiripple filter useful for interpolating input signals. `n` is the filter order and it must be an integer. `l`, also an integer, is the interpolation factor. `alpha` is the bandlimitedness factor, identical to the same feature in `intfilt`.

`alpha` is inversely proportional to the transition bandwidth of the filter. It also affects the bandwith of the don't-care regions in the stopband. Specifying `alpha` allows you to control how much of the Nyquist interval your input signal occupies. This can be beneficial for signals to be interpolated because it allows you to increase the transition band width without affecting the interpolation, resulting in better stopband attenuation for a given `l`. If you set `alpha` to 1, `fireqint` assumes that your signal occupies the entire Nyquist interval. Setting `alpha` to a value less than one allows for don't-care regions in the stopband. For example, if your input occupies half the Nyquist interval, you could set `alpha` to 0.5.

The signal to be interpolated is assumed to have zero (or negligible) power in the frequency band between ($alpha*\pi$) and π. `alpha` must therefore be a positive scalar between 0 and 1. `fireqint` treat such bands as don't-care regions for assessing filter design.

`b = fireqint(n,l,alpha,w)` allows you to specify a vector of weights in `w`. The number of weights required in `w` is given by `1 + floor(l/2)`. The weights in `w` are applied to the passband ripple and stopband attenuations. Using weights (values between 0 and 1) enables you to specify different attenuations in different parts of the stopband, as well as providing the ability to adjust the compromise between passband ripple and stopband attenuation.

`b = fireqint('minorder',l,alpha,r)` allows you to design a minimum-order filter that meets the design specifications. `r` is a vector

of maximum deviations or ripples from the ideal filter magnitude response. When you use the input argument **minorder**, you must provide the vector r. The number of elements required in r is given by $1 + \texttt{floor(l/2)}$.

b = fireqint({'**minorder**',initord},l,alpha,r) adds the argument initord so you can provide an initial estimate of the filter order. Any positive integer is valid here. Again, you must provide r, the vector of maximum deviations or ripples, from the ideal filter magnitude response.

Examples   Design a minimum order interpolation filter for l = 6 and alpha = 0.8. A vector of ripples must be supplied with the input argument **minorder**.

```
b = fireqint('minorder',6,.8,[0.01 .1 .05 .02]);
hm = mfilt.firinterp(6,b); % Create a polyphase interpolator filter
zerophase(hm);
```

Here is the resulting plot of the zerophase response for hm.

For hm, the minimum order filter with the requested design specifications, here is the filter information.

```
hm =
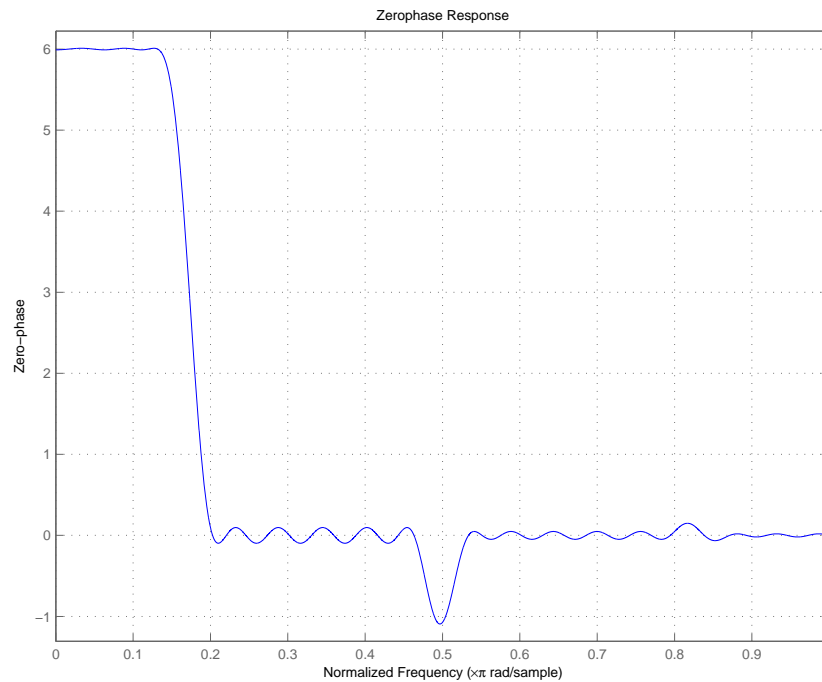
FilterStructure: 'Direct-Form FIR Polyphase Interpolator'
        Arithmetic: 'double'
         Numerator: [1x70 double]
InterpolationFactor: 6
  PersistentMemory: false
```

See Also firgr | firhalfband | firls | firnyquist | mfilt | intfilt

Purpose   Constrained equiripple FIR filter

Syntax
```
B = firceqrip(n,Fo,DEV)
B = firceqrip(...,'slope',r)
B = firceqrip(...,'passedge')
B = firceqrip(...,'stopedge')
B = firceqrip(...,'high')
B = firceqrip(...,'min')
B = firceqrip(...,'invsinc',C)
B = firceqrip(...,'invdiric',C)
```

Description   `B = firceqrip(n,Fo,DEV)` designs an order n filter (filter length equal n + 1) lowpass FIR filter with linear phase.

`firceqrip` produces the same equiripple lowpass filters that `firpm` produces using the Parks-McClellan algorithm. The difference is how you specify the filter characteristics for the function.

The input argument `Fo` specifies the frequency at the upper edge of the passband in normalized frequency (0<Fo<1). The two-element vector `dev` specifies the peak or maximum error allowed in the passband and stopbands. Enter `[d1 d2]` for `dev` where `d1` sets the passband error and `d2` sets the stopband error.

`B = firceqrip(...,'slope',r)` uses the input keyword `'slope'` and input argument `r` to design a filter with a nonequiripple stopband. `r` is specified as a positive constant and determines the slope of the stopband attenuation in dB/normalized frequency. Greater values of `r` result in increased stopband attenuation in dB/normalized frequency.

`B = firceqrip(...,'passedge')` designs a filter where `Fo` specifies the frequency at which the passband starts to rolloff.

`B = firceqrip(...,'stopedge')` designs a filter where `Fo` specifies the frequency at which the stopband begins.

`B = firceqrip(...,'high')` designs a high pass FIR filter instead of a lowpass filter.

`B = firceqrip(...,'min')` designs a minimum-phase filter.

B = firceqrip(...,'invsinc',C) designs a lowpass filter whose magnitude response has the shape of an inverse sinc function. This may be used to compensate for sinc-like responses in the frequency domain such as the effect of the zero-order hold in a D/A converter. The amount of compensation in the passband is controlled by C, which is specified as a scalar or two-element vector. The elements of C are specified as follows:

- If C is supplied as a real-valued scalar or the first element of a two-element vector, firceqrip constructs a filter with a magnitude response of 1/sinc(C*pi*F) where F is the normalized frequency.

- If C is supplied as a two-element vector, the inverse-sinc shaped magnitude response is raised to the positive power C(2). If we set P=C(2), firceqrip constructs a filter with a magnitude response 1/sinc(C*pi*F)^P.

If this FIR filter is used with a cascaded integrator-comb (CIC) filter, setting C(2) equal to the number of stages compensates for the multiplicative effect of the successive sinc-like responses of the CIC filters.

Note Since the value of the inverse sinc function becomes unbounded at C=1/F, the value of C should be greater the reciprocal of the passband edge frequency. This can be expressed as Fo<1/C. For users familar with CIC decimators, C is equal to 1/2 the product of the differential delay and decimation factor.

B = firceqrip(...,'invdiric',C) designs a lowpass filter with a passband that has the shape of an inverse Dirichlet sinc function. The frequency response of the inverse Dirichlet sinc function is given by

$$\{rC(\frac{\sin(f/2r)}{\sin(Cf/2)}\}^p$$

where *C*, *r*, and *p* are scalars. The input C can be a scalar or vector containing 2 or 3 elements. If C is a scalar, p and r equal 1. If C is a two-element vector, the first element is C and the second element is p, [C p]. If C is a three-element vector, the third element is r, [C p r].

Examples   To introduce a few of the variations on FIR filters that you design with firceqrip, these five examples cover both the default syntax b = firceqrip(n,wo,del) and some of the optional input arguments. For each example, the input arguments n, wo, and del remain the same.

Example 1

Design an order = 30 FIR filter.

```
b = firceqrip(30,0.4,[0.05 0.03]); fvtool(h)
```

When the plot appears in the Filter Visualization Tool window, select **Analysis > Overlay Analysis > Phase Response**. Then select **View > Full View**. This displays the following plot.

Example 2

Design an order = 30 FIR filter with the stopedge keyword to define the response at the edge of the filter stopband.

```
b = firceqrip(30,0.4,[0.05 0.03],'stopedge'); fvtool(h)
```

Example 3

Design an order = 30 FIR filter with the slope keyword and r = 20.

```
b = firceqrip(30,0.4,[0.05 0.03],'slope',20,'stopedge'); fvtool(h)
```

Example 4

Design an order = 30 FIR filter defining the stopband and specifying that the resulting filter is minimum phase with the **min** keyword.

```
b = firceqrip(30,0.4,[0.05 0.03],'stopedge','min'); fvtool(h)
```

Comparing this filter to the filter in Example 1, the cutoff frequency wo = 0.4 now applies to the edge of the stopband rather than the point at which the frequency response magnitude is 0.5.

Viewing the zero-pole plot shown here reveals this is a minimum phase FIR filter — the zeros lie on or inside the unit circle, z = 1.

Pole/Zero Plot

Example 5

Design an order = 30 FIR filter with the invsinc keyword to shape the filter passband with an inverse sinc function.

```
b = firceqrip(30,0.4,[0.05 0.03],'invsinc',[2 1.5]); fvtool(h)
```

With the inverse sinc function being applied defined as $1/\text{sinc}(2\text{*w})^{1.5}$, the figure shows the reshaping of the passband that results from using the **invsinc** keyword option, and entering c as the two-element vector [2 1.5].

Inverse-Dirichlet-Sinc–Shaped Passband

Design two order 30 constrained equiripple FIR filters with inverse-Dirichlet-sinc-shaped passbands. The cutoff frequency in both designs is π/4 radians/sample. Set C=1 in one design C=2 in the second design. The maximum passband and stopband ripple is 0.05. Set p=1 in one design and p=2 in the second design.

Design the filters.

```
b1 = firceqrip(30,0.25,[0.05 0.05],'invdiric',[1 1]);
b2 = firceqrip(30,0.25,[0.05 0.05],'invdiric',[2 2]);
```

Obtain the filter frequency responses using freqz. Plot the magnitude responses.

```
[h1,~] = freqz(b1,1);
[h2,w] = freqz(b2,1);
plot(w,abs(h1)); hold on;
plot(w,abs(h2),'r');
axis([0 pi 0 1.5]);
xlabel('Radians/sample');
```

```
ylabel('Magnitude');
legend('C=1 p=1','C=2 p=2');
```

Inspect the stopband ripple in the design with C=1 and p=1. The constrained design sets the maximum ripple to be 0.05. Zoom in on the stopband from the cutoff frequency of π/4 radians/sample to 3π/4 radians/sample.

```
figure;
plot(w,abs(h1));
set(gca,'xlim',[pi/4 3*pi/4]);
grid on;
```

See Also   diric | fdesign.decimator | firhalfband | firnyquist | firgr | ifir | iirgrpdelay | iirlpnorm | iirlpnormc | fircls | firls | firpm | sinc

fircls

Purpose   FIR Constrained Least Squares filter

Syntax
```
hd = design(d,'fircls')
hd = design(d,'fircls','FilterStructure',value)
hd = design(d,'fircls','PassbandOffset',value)
hd = design(d,'fircls','zerophase',value)
```

Description `hd = design(d,'fircls')` designs a FIR Constrained Least Squares (CLS) filter, `hd`, from a filter specifications object, `d`.

`hd = design(d,'fircls','FilterStructure',value)` where `value` is one of the following filter structures:

- `'dffir'`, a discrete-time, direct-form FIR filter (the default value)

- `'dffirt'`, a discrete-time, direct-form FIR transposed filter

- `'dfsymfir'`, a discrete-time, direct-form symmetric FIR filter

- `'fftfir'`, a discrete-time, overlap-add, FIR filter

`hd = design(d,'fircls','PassbandOffset',value)` where `value` sets the passband band gain in dB. The `PassbandOffset` and `Ap` values affect the upper and lower approximation bound in the passband as follows:

- Lower bound = (`PassbandOffset`-Ap/2)

- Upper bound = (`PassbandOffset`+A/2)

For bandstop filters, the `PassbandOffset` is a vector of length two that specifies the first and second passband gains. The `PassbandOffset` value defaults to 0 for lowpass, highpass and bandpass filters. The `PassbandOffset` value defaults to [0 0] for bandstop filters.

`hd = design(d,'fircls','zerophase',value)` where `value` is either `'true'` ('1') or `'false'` ('0') . If `zerophase` is true, the lower approximation bound in the stopband is forced to zero (i.e., the filter has a zero phase response). Zerophase is false (0) by default.

To determine the available design options, use designopts with the
specification object and the design method as input arguments as
shown.

```
designopts(d,'fircls')
```

For complete help about using fircls, refer to the command line help
system. For example, to get specific information about using fircls
with d, the specification object, enter the following at the MATLAB
prompt.

```
help(d,'fircls')
```

Examples

The following example designs a constrained least-squares FIR lowpass
filter.

```
h  = fdesign.lowpass('n,fc,ap,ast', 50, 0.3, 2, 30);
Hd = design(h, 'fircls');
fvtool(Hd)
```


The following example constructs a constrained least-squares FIR
bandpass filter.

```
d = fdesign.bandpass('N,Fc1,Fc2,Ast1,Ap,Ast2',...
```

```
30,0.4,0.6,60,1,60);
Hd = design(d, 'fircls');
fvtool(Hd)
```


See Also   cheby1 | cheby2 | ellip

Purpose	Parks-McClellan FIR filter

Syntax
```
b = firgr(n,f,a,w)
b = firgr(n,f,a,'hilbert')
b = firgr(m,f,a,r),
b = firgr({m,ni},f,a,r)
b = firgr(n,f,a,w,e)
b = firgr(n,f,a,s)
b = firgr(n,f,a,s,w,e)
b = firgr(...,'1')
b = firgr(...,'minphase')
b = firgr(..., 'check')
b = firgr(...,{lgrid}),
[b,err] = firgr(...)
[b,err,res] = firgr(...)
b = firgr(n,f,fresp,w)
b = firgr(n,f,{fresp,p1,p2,...},w)
b = firgr(n,f,a,w)
```

Description   firgr is a minimax filter design algorithm you use to design the following types of real FIR filters:

- Types 1-4 linear phase:

  - Type 1 is even order, symmetric

  - Type 2 is odd order, symmetric

  - Type 3 is even order, antisymmetric

  - Type 4 is odd order, antisymmetric

- Minimum phase

- Maximum phase

- Minimum order (even or odd)

- Extra ripple

- Maximal ripple

- Constrained ripple

- Single-point band (notching and peaking)

- Forced gain

- Arbitrary shape frequency response curve filters

`b = firgr(n,f,a,w)` returns a length n+1 linear phase FIR filter which has the best approximation to the desired frequency response described by `f` and `a` in the minimax sense. `w` is a vector of weights, one per band. When you omit `w`, all bands are weighted equally. For more information on the input arguments, refer to `firpm` in *Signal Processing Toolbox User's Guide*.

`b = firgr(n,f,a,'hilbert')` and `b = firgr(n,f,a,'differentiator')` design FIR Hilbert transformers and differentiators. For more information on designing these filters, refer to `firpm` in *Signal Processing Toolbox User's Guide*.

`b = firgr(m,f,a,r)`, where m is one of 'minorder', 'mineven' or 'minodd', designs filters repeatedly until the minimum order filter, as specified in `m`, that meets the specifications is found. `r` is a vector containing the peak ripple per frequency band. You must specify `r`. When you specify 'mineven' or 'minodd', the minimum even or odd order filter is found.

`b = firgr({m,ni},f,a,r)` where m is one of 'minorder', 'mineven' or 'minodd', uses `ni` as the initial estimate of the filter order. `ni` is optional for common filter designs, but it must be specified for designs in which `firpmord` cannot be used, such as while designing differentiators or Hilbert transformers.

`b = firgr(n,f,a,w,e)` specifies independent approximation errors for different bands. Use this syntax to design extra ripple or maximal ripple filters. These filters have interesting properties such as having the minimum transition width. `e` is a cell array of strings specifying the approximation errors to use. Its length must equal the number of bands. Entries of `e` must be in the form 'e#' where # indicates which approximation error to use for the corresponding band. For example, when `e = {'e1','e2','e1'}`, the first and third bands use the same

approximation error `'e1'` and the second band uses a different one `'e2'`. Note that when all bands use the same approximation error, such as `{'e1','e1','e1',...}`, it is equivalent to omitting e, as in `b = firgr(n,f,a,w)`.

`b = firgr(n,f,a,s)` is used to design filters with special properties at certain frequency points. `s` is a cell array of strings and must be the same length as `f` and `a`. Entries of `s` must be one of:

- `'n'` — normal frequency point.

- `'s'` — single-point band. The frequency "band" is given by a single point. The corresponding gain at this frequency point must be specified in `a`.

- `'f'` — forced frequency point. Forces the gain at the specified frequency band to be the value specified.

- `'i'` — indeterminate frequency point. Use this argument when adjacent bands abut one another (no transition region).

For example, the following command designs a bandstop filter with zero-valued single-point stop bands (notches) at 0.25 and 0.55.

```
b = firgr(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],...
[1 1 0 1 1 0 1 1],{'n' 'n' 's' 'n' 'n' 's' 'n' 'n'})
```

`b = firgr(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],...{'n' 'i' 'f' 'n' 'n' 'n'})` designs a highpass filter with the gain at 0.06 forced to be zero. The band edge at 0.055 is indeterminate since the first two bands actually touch. The other band edges are normal.

`b = firgr(n,f,a,s,w,e)` specifies weights and independent approximation errors for filters with special properties. The weights and properties are included in vectors `w` and `e`. Sometimes, you may need to use independent approximation errors to get designs with forced values to converge. For example,

```
b = firgr(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'}, [10 1 1] ,{'e1' 'e2' 'e3'});
```

`b = firgr(...,'1')` designs a type 1 filter (even-order symmetric). You can specify type 2 (odd-order symmetric), type 3 (even-order antisymmetric), and type 4 (odd-order antisymmetric) filters as well. Note that restrictions apply to a at f = 0 or f = 1 for FIR filter types 2, 3, and 4.

`b = firgr(...,'minphase')` designs a minimum-phase FIR filter. You can use the argument `'maxphase'` to design a maximum phase FIR filter.

`b = firgr(..., 'check')` returns a warning when there are potential transition-region anomalies.

`b = firgr(...,{lgrid})`, where {lgrid} is a scalar cell array. The value of the scalar controls the density of the frequency grid by setting the number of samples used along the frequency axis.

`[b,err] = firgr(...)` returns the unweighted approximation error magnitudes. `err` contains one element for each independent approximation error returned by the function.

`[b,err,res] = firgr(...)` returns the structure `res` comprising optional results computed by `firgr`. `res` contains the following fields.

Structure Field	Contents
res.fgrid	Vector containing the frequency grid used in the filter design optimization
res.des	Desired response on fgrid
res.wt	Weights on fgrid
res.h	Actual frequency response on the frequency grid
res.error	Error at each point (desired response - actual response) on the frequency grid
res.iextr	Vector of indices into fgrid of external frequencies
res.fextr	Vector of external frequencies

Structure Field	Contents
res.order	Filter order
res.edgecheck	Transition-region anomaly check. One element per band edge. Element values have the following meanings: 1 = OK, 0 = probable transition-region anomaly, -1 = edge not checked. Computed when you specify the 'check' input option in the function syntax.
res.iterations	Number of s iterations for the optimization
res.evals	Number of function evaluations for the optimization

firgr is also a "function function," allowing you to write a function that defines the desired frequency response.

b = firgr(n,f,fresp,w) returns a length N + 1 FIR filter which has the best approximation to the desired frequency response as returned by the user-defined function fresp. Use the following firgr syntax to call fresp:

[dh,dw] = fresp(n,f,gf,w)

where:

- fresp is the string variable that identifies the function that you use to define your desired filter frequency response.

- n is the filter order.

- f is the vector of frequency band edges which must appear monotonically between 0 and 1, where 1 is one-half of the sampling frequency. The frequency bands span f(k) to f(k+1) for k odd. The intervals f(k+1) to f(k+2) for k odd are "transition bands" or "don't care" regions during optimization.

firgr

- gf is a vector of grid points that have been chosen over each specified frequency band by firgr, and determines the frequencies at which firgr evaluates the response function.

- w is a vector of real, positive weights, one per band, for use during optimization. w is optional in the call to firgr. If you do not specify w, it is set to unity weighting before being passed to fresp.

- dh and dw are the desired frequency response and optimization weight vectors, evaluated at each frequency in grid gf.

firgr includes a predefined frequency response function named 'firpmfrf2'. You can write your own based on the simpler 'firpmfrf'. See the help for private/firpmfrf for more information.

b = firgr(n,f,{*fresp*,p1,p2,...},w) specifies optional arguments p1, p2,..., pn to be passed to the response function fresp.

b = firgr(n,f,a,w) is a synonym for b = firgr(n,f,{'firpmfrf2',a},w), where a is a vector containing your specified response amplitudes at each band edge in f. By default, firgr designs symmetric (even) FIR filters. 'firpmfrf2' is the predefined frequency response function. If you do not specify your own frequency response function (the fresp string variable), firgr uses 'firpmfrf2'.

b = firgr(...,'h') and b = firgr(...,'d') design antisymmetric (odd) filters. When you omit the 'h' or 'd' arguments from the firgr command syntax, each frequency response function fresp can tell firgr to design either an even or odd filter. Use the command syntax sym = *fresp*('defaults',{n,f,[],w,p1,p2,...}).

firgr expects fresp to return sym = 'even' or sym = 'odd'. If fresp does not support this call, firgr assumes even symmetry.

For more information about the input arguments to firgr, refer to firpm.

Examples These examples demonstrate some filters you might design using firgr.

Example 1

design an FIR filter with two single-band notches at 0.25 and 0.55

```
b1 = firgr(42,[0 0.2 0.25 0.3 0.5 0.55 0.6 1],[1 1 0 1 1 0 1 1],...
{'n' 'n' 's' 'n' 'n' 's' 'n' 'n'});
```

Example 2

design a highpass filter whose gain at 0.06 is forced to be zero. The gain at 0.055 is indeterminate since it should abut the band.

```
b2 = firgr(82,[0 0.055 0.06 0.1 0.15 1],[0 0 0 0 1 1],...
{'n' 'i' 'f' 'n' 'n' 'n'});
```

Example 3

design a second highpass filter with forced values and independent approximation errors.

```
b3 = firgr(82,[0 0.055 0.06 0.1 0.15 1], [0 0 0 0 1 1], ...
{'n' 'i' 'f' 'n' 'n' 'n'}, [10 1 1] ,{'e1' 'e2' 'e3'});
```

Use the filter visualization tool to view the results of the filters created in these examples.

```
fvtool(b1,1,b2,1,b3,1)
```

Here is the figure from FVTool.

Magnitude Response (dB)

References   Shpak, D.J. and A. Antoniou, "A generalized Remez method for the design of FIR digital filters," *IEEE Trans. Circuits and Systems*, pp. 161-174, Feb. 1990.

See Also   butter | cheby1 | cheby2 | ellip | freqz | filter | firls | fircls | firpm

Purpose   Halfband FIR filter

Syntax   b = firhalfband(n,fp)
                   b = firhalfband(n,win)
                   b = firhalfband(n,dev,'**dev**')
                   b = firhalfband('**minorder**',fp,dev)
                   b = firhalfband('**minorder**',fp,dev,'**kaiser**')
                   b = firhalfband(...,'**high**')
                   b = firhalfband(...,'**minphase**')

Description   b = firhalfband(n,fp) designs a lowpass halfband FIR filter of order
                   n with an equiripple characteristic. n must be an even integer. fp
                   determines the passband edge frequency, and it must satisfy $0 < fp <$
                   $1/2$, where $1/2$ corresponds to $\pi/2$ rad/sample.

                   b = firhalfband(n,win) designs a lowpass Nth-order filter using the
                   truncated, windowed-impulse response method instead of the equiripple
                   method. win is an n+1 length vector. The ideal impulse response is
                   truncated to length n + 1, and then multiplied point-by-point with the
                   window specified in win.

                   b = firhalfband(n,dev,'**dev**') designs an Nth-order lowpass
                   halfband filter with an equiripple characteristic. Input argument dev
                   sets the value for the maximum passband and stopband ripple allowed.

                   b = firhalfband('**minorder**',fp,dev) designs a lowpass
                   minimum-order filter, with passband edge fp. The peak ripple is
                   constrained by the scalar dev. This design uses the equiripple method.

                   b = firhalfband('**minorder**',fp,dev,'**kaiser**') designs a lowpass
                   minimum-order filter, with passband edge fp. The peak ripple is
                   constrained by the scalar dev. This design uses the Kaiser window
                   method.

                   b = firhalfband(...,'**high**') returns a highpass halfband FIR filter.

                   b = firhalfband(...,'**minphase**') designs a minimum-phase FIR
                   filter such that the filter is a spectral factor of a halfband filter
                   (recall that h = conv(b,fliplr(b)) is a halfband filter). This can
                   be useful for designing perfect reconstruction, two-channel FIR filter

banks. The **minphase** option for `firhalfband` is not available for the window-based halfband filter designs — `b = firhalfband(n,win)` and `b = firhalfband('`**minorder** `',fp,dev,'`**kaiser**`')`.

In the minimum phase cases, the filter order must be odd.

Examples

This example designs a minimum order halfband filter with specified maximum ripple:

```
b = firhalfband('minorder',.45,0.0001);
h = dfilt.dfsymfir(b);
impz(b) % Impulse response is zero for every other sample
```


The next example designs a halfband filter with specified maximum ripple of 0.0001 dB in the pass and stop bands.

```
b = firhalfband(98,0.0001,'dev');
h = mfilt.firdecim(2,b); % Create a polyphase decimator
freqz(h); % 80 dB attenuation in the stopband
```

References   Saramaki, T, "Finite Impulse Response Filter Design," *Handbook for Digital Signal Processing.* S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

See Also   firnyquist | firgr | fir1 | firls | firpm

firlp2lp

Purpose	Convert FIR Type I lowpass to FIR Type 1 lowpass with inverse bandwidth
Syntax	`g = firlp2lp(b)`
Description	`g = firlp2lp(b)` transforms the Type I lowpass FIR filter `b` with zero-phase response Hr(w) to a Type I lowpass FIR filter `g` with zero-phase response [1 - Hr(π-w)].
	When `b` is a narrowband filter, `g` will be a wideband filter and vice versa. The passband and stopband ripples of `g` will be equal to the stopband and passband ripples of `b`.
Examples	Overlay the original narrowband lowpass and the resulting wideband lowpass

```
b = firgr(36,[0 .2 .25 1],[1 1 0 0],[1 5]);
zerophase(b);
hold on
h = firlp2lp(b);
zerophase(h);
```

References	Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*. S.K. Mitra and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.
See Also	`firlp2hp` | `zerophase`

Purpose	Convert FIR lowpass filter to Type I FIR highpass filter
Syntax	`g = firlp2hp(b)`
	`g = firlp2hp(b,'narrow')`
	`g = firlp2hp(b,'wide')`

Description   `g = firlp2hp(b)` transforms the lowpass FIR filter `b` into a Type I highpass FIR filter `g` with zero-phase response Hr(π-w). Filter `b` can be any FIR filter, including a nonlinear-phase filter.

The passband and stopband ripples of `g` will be equal to the passband and stopband ripples of `b`.

`g = firlp2hp(b,'narrow')` transforms the lowpass FIR filter `b` into a Type I narrow band highpass FIR filter `g` with zero-phase response Hr(π-w). `b` can be any FIR filter, including a nonlinear-phase filter.

`g = firlp2hp(b,'wide')` transforms the Type I lowpass FIR filter `b` with zero-phase response Hr(w) into a Type I wide band highpass FIR filter `g` with zero-phase response 1 - Hr(w). Note the restriction that b must be a Type I linear-phase filter.

For this case, the passband and stopband ripples of `g` will be equal to the stopband and passband ripples of `b`.

Examples   Overlay the original narrowband lowpass (the prototype filter) and the post-conversion narrowband highpass and wideband highpass filters to compare and assess the conversion. The following plot shows the results.

```
b = firgr(36,[0 .2 .25 1],[1 1 0 0],[1 3]);
h = firlp2hp(b);
hfvt = fvtool(b,1,h,1);
legend(hfvt,'Prototype Lowpass','Narrowband Highpass');
```

firlp2hp

References   Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

See Also   `firlp2lp | zerophase`

Purpose   Least P-norm optimal FIR filter

Syntax   b = firlpnorm(n,f,edges,a)
                    b = firlpnorm(n,f,edges,a,w)
                    b = firlpnorm(n,f,edges,a,w,p)
                    b = firlpnorm(n,f,edges,a,w,p,dens)
                    b = firlpnorm(n,f,edges,a,w,p,dens,initnum)
                    b = firlpnorm(...,'minphase')
                    [b,err] = firlpnorm(...)

Description   b = firlpnorm(n,f,edges,a) returns a filter of numerator order n
                    which represents the best approximation to the frequency response
                    described by f and a in the least-Pth norm sense. P is set to 128 by
                    default, which essentially equivalent to the infinity norm. Vector edges
                    specifies the band-edge frequencies for multiband designs. firlpnorm
                    uses an unconstrained quasi-Newton algorithm to design the specified
                    filter.

                    f and a must have the same number of elements, which can exceed the
                    number of elements in edges. This lets you specify filters with any gain
                    contour within each band. However, the frequencies in edges must also
                    be in vector f. Always use freqz to check the resulting filter.

                    Note firlpnorm uses a nonlinear optimization routine that may not
                    converge in some filter design cases. Furthermore the algorithm is not
                    well-suited for certain large-order (order > 100) filter designs.

                    ---

                    b = firlpnorm(n,f,edges,a,w) uses the weights in w to weight the
                    error. w has one entry per frequency point (the same length as f and
                    a) which tells firlpnorm how much emphasis to put on minimizing
                    the error in the vicinity of each frequency point relative to the other
                    points. For example,

                    b = firlpnorm(20,[0 .15 .4 .5 1],[0 .4 .5 1],...
                    [1 1.6 1 0 0],[1 1 1 10 10])

firlpnorm

designs a lowpass filter with a peak of 1.6 within the passband, and with emphasis placed on minimizing the error in the stopband.

`b = firlpnorm(n,f,edges,a,w,p)` where `p` is a two-element vector `[pmin pmax]` lets you specify the minimum and maximum values of `p` used in the least-pth algorithm. Default is `[2 128]` which essentially yields the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even numbers. The design algorithm starts optimizing the filter with `pmin` and moves toward an optimal filter in the `pmax` sense. When `p` is the string '**inspect**', `firlpnorm` does not optimize the resulting filter. You might use this feature to inspect the initial zero placement.

`b = firlpnorm(n,f,edges,a,w,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is `[dens*(n+1)]`. The default is 20. You can specify `dens` as a single-element cell array. The grid is equally spaced.

`b = firlpnorm(n,f,edges,a,w,p,dens,initnum)` lets you determine the initial estimate of the filter numerator coefficients in vector `initnum`. This can prove helpful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initnum`.

`b = firlpnorm(...,'minphase')` where string 'minphase' is the last argument in the argument list generates a minimum-phase FIR filter. By default, `firlpnorm` design mixed-phase filters. Specifying input option 'minphase' causes `firlpnorm` to use a different optimization method to design the minimum-phase filter. As a result of the different optimization used, the minimum-phase filter can yield slightly different results.

`[b,err] = firlpnorm(...)` returns the least-pth approximation error `err`.

Examples   To demonstrate `firlpnorm`, here are two examples — the first designs a lowpass filter and the second a highpass, minimum-phase filter.

```
% Lowpass filter with a peak of 1.4 in the passband.
b = firlpnorm(22,[0 .15 .4 .5 1],[0 .4 .5 1],[1 1.4 1 0 0],...
```

```
[1 1 1 2 2]);
fvtool(b)
```

From the figure you see the resulting filter is lowpass, with the desired 1.4 peak in the passband (notice the 1.4 specified in vector a).

Now for the minimum-phase filter.

```
% Highpass minimum-phase filter optimized for the 4-norm.
b = firlpnorm(44,[0 .4 .45 1],[0 .4 .45 1],[0 0 1 1],[5 1 1 1],...
[2 4],'minphase');
fvtool(b)
```

As shown in the next figure, this is a minimum-phase, highpass filter.

firlpnorm

The next zero-pole plot shows the minimum phase nature more clearly.

Zero–Pole Plot of Minimum Phase FIR Filter

References   Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing* Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

See Also   firgr | iirgrpdelay | iirlpnorm | iirlpnormc | filter | fvtool | freqz | zplane

Purpose　　　Least-square linear-phase FIR filter design

Syntax　　　```
b = firls(n,f,a)
b = firls(n,f,a,w)
b = firls(n,f,a,ftype)
b = firls(n,f,a,w,ftype)
```
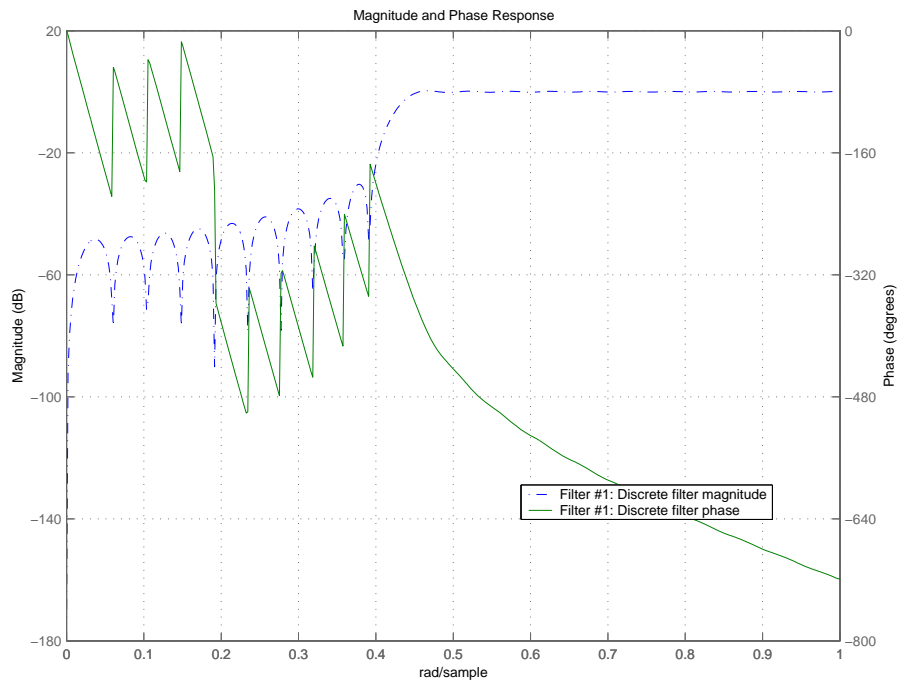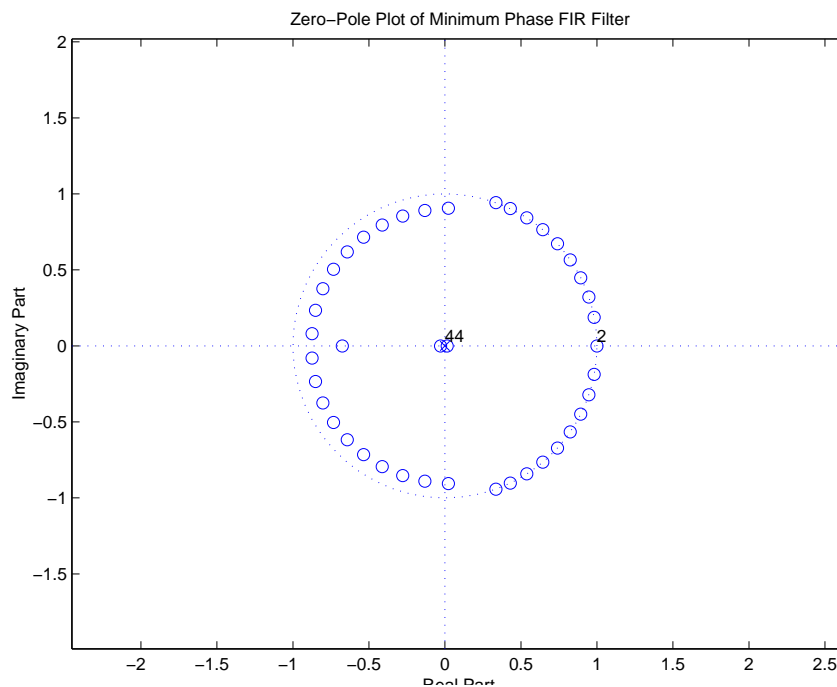
Description　　　`b = firls(n,f,a)` returns row vector `b` containing the n+1 coefficients of the order n FIR filter. This filter has frequency-amplitude characteristics approximately matching those given by vectors, `f` and `a`.

`b = firls(n,f,a,w)` uses the weights in vector `w`, to weigh the error.

`b = firls(n,f,a,ftype)` specifies a filter type where `ftype` is:

- `'hilbert'`

- `'differentiator'`

`b = firls(n,f,a,w,ftype)` uses the weights in vector `w` to weigh the error. It also specifies a filter type where `ftype` is:

- `'hilbert'`

- `'differentiator'`

Input Arguments

n - Filter order

(default) | integer scalar

Order of the filter, specified as an integer scalar. For odd orders, the frequency response at the Nyquist frequency is necessarily 0. For this reason, `firls` always uses an even filter order for configurations with a passband at the Nyquist frequency. If you specify an odd-valued n, `firls` increments it by 1.

Example: 8

Data Types
int8 | int16 | int32 | int64
Complex Number Support: No

f - Pairs of frequency points
vector of numeric values

Pairs of frequency points, specified as a vector of values ranging between 0 and 1, where 1 corresponds to the Nyquist frequency. The frequencies must be in increasing order, and duplicate frequency points are allowed. You can use duplicate frequency points to design filters exactly like those returned by the fir1 and fir2 functions with a rectangular (rectwin) window.

f and a are the same length. This length must be an even number.

Example: [0 0.3 0.4 1]

Data Types
double | single
Complex Number Support: No

a - Amplitude values
vector of numeric values

Amplitude values of the function at each frequency point, specified as a vector of the same length as f. This length must be an even number.

The desired amplitude at frequencies between pairs of points (*f*(*k*), *f*(*k*+1)) for *k* odd, is the line segment connecting the points (*f*(*k*), *a*(*k*)) and (*f*(*k*+1), *a*(*k*+1)).

The desired amplitude at frequencies between pairs of points (*f*(*k*), *f*(*k*+1)) for *k* even is unspecified. These are transition or "don't care" regions.

Example: [1 1 0 0]

Data Types
double | single
Complex Number Support: No

w - Weights
vector of numeric values

Weights to weigh the fit for each frequency band, specified as a vector of length half the length of f and a, so there is exactly one weight per band. w indicates how much emphasis to put on minimizing the integral squared error in each band, relative to the other bands.

Example: [0.5 1]

Data Types
double | single
Complex Number Support: No

ftype - Filter type
'hilbert' and 'differentiator'

Filter type, specified as either `hilbert' or `differentiator'.

Example: 'hilbert'

Data Types
char

Output Arguments

b - Filter coefficients
vector of numeric values

Filter coefficients, returned as a numeric vector of n+1 values, where n is the filter order.

b = firls(n,f,a) designs a linear-phase filter of type I (n odd) and type II (n) . The output coefficients, or "taps," in b obey the relation:

$$b(k) = b(n+2-k), k = 1, \ldots , n + 1$$

b = firls(n,f,a,'hilbert') designs a linear-phase filter with odd symmetry (type III and type IV). The output coefficients, or "taps," in b obey the relation:

firls

$$b(k) = -b(n+2\text{-}k), \ k = 1, \ ... \ , \ n + 1$$

`b = firls(n,f,a,'differentiator')` designs type III and type IV filters, using a special weighting technique. For nonzero amplitude bands, the integrated squared error has a weight of $(1/f)^2$. This weighting causes the error at low frequencies to be much smaller than at high frequencies. For FIR differentiators, which have an amplitude characteristic proportional to frequency, the filters minimize the relative integrated squared error. This value is the integral of the square of the ratio of the error to the desired amplitude.

Examples

Design a Lowpass Filter with Transition Band

The following illustrates how to design a lowpass filter of order 225 with transition band.

Create the frequency and amplitude vectors, `f` and `a`.

```
f = [0 0.25 0.3 1]
a = [1 1 0 0]
```

Use `firls` to obtain the `n+1` coefficients of the order `n` lowpass FIR filter.

```
b = firls(255,f,a);
```

Design an Antisymmetric Filter with Piecewise Linear Passbands

The following shows how to design a 24th-order anti-symmetric filter with piecewise linear passbands, and plot the desired and actual amplitude responses.

Create the frequency and amplitude vectors, `f` and `a`.

```
f = [0 0.3 0.4 0.6 0.7 0.9];
a = [0 1 0 0 0.5 0.5];
```

Use `firls` to obtain the 25 coefficients of the filter.

```
b = firls(24,f,a,'hilbert');
```

Plot the ideal amplitude response along with the transition regions.

```
plot(f.*pi,a,'o','markerfacecolor',[1 0 0]);
hold on;
plot(f.*pi,a,'r--','linewidth',2);
```

Use freqz to obtain the frequency response of the designed filter and
plot the magnitude response of the filter.

```
[H,F] = freqz(b,1);
plot(F,abs(H));
set(gca,'xlim',[O pi])
legend('Filter Specification','Transition Regions','Magnitude Response
```

firls
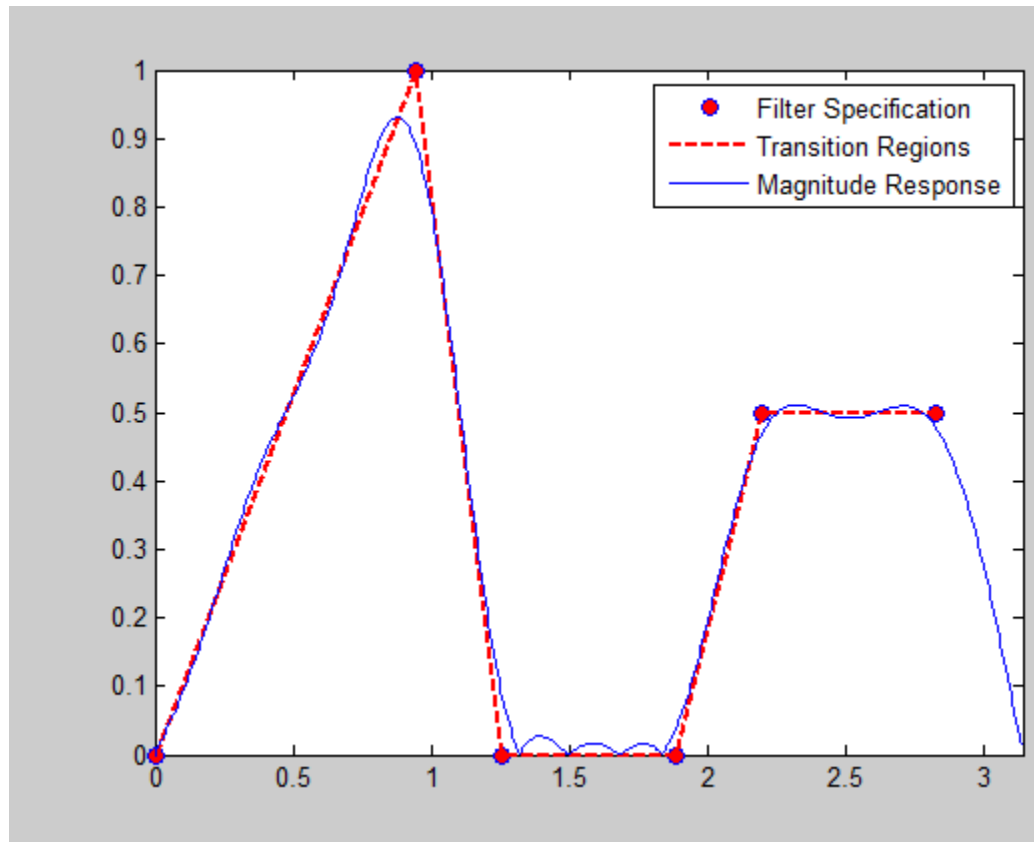
Algorithms firls designs a linear-phase FIR filter. This filter minimizes the weighted, integrated squared error between an ideal piecewise linear function and the magnitude response of the filter over a set of desired frequency bands.

Reference [1] describes the theoretical approach behind firls. The function solves a system of linear equations involving an inner product matrix of size roughly n/2 using the MATLAB \ operator.

This function designs type I, II, III, and IV linear-phase filters. Type I and II are the defaults for n even and odd respectively. The `'hilbert'` and `'differentiator'` flags produce type III (n even) and IV (n odd) filters. The various filter types have different symmetries and constraints on their frequency responses (see [2] for details).

Linear Phase Filter Type	Filter Order	Symmetry of Coefficients	Response H(f), f = 0	Response H(f), f = 1 (Nyquist)
Type I	Even	b(k) = b(n+2-k), k=1,..., n+1	No restriction	No restriction
Type II	Even	b(k) = b(n+2-k), k=1,..., n+1	No restriction	$H(1) = 0$
Type III	Odd	b(k) = –b(n+2-k), k=1,..., n+1	$H(0) = 0$	$H(1) = 0$
Type IV	Odd	b(k) = –b(n+2-k), k=1,..., n+1	$H(0) = 0$	No restriction

Definitions

Diagnostics

Error and warning messages

One of the following diagnostic messages is displayed when an incorrect argument is used:

```
F must be even length.
F and A must be equal lengths.
Requires symmetry to be 'hilbert' or 'differentiator'.
Requires one weight per band.
Frequencies in F must be nondecreasing.
Frequencies in F must be in range [0,1].
```

A more serious warning message is

```
Warning: Matrix is close to singular or badly scaled.
```

This tends to happen when the product of the filter length and transition width grows large. In this case, the filter coefficients b might not represent the desired filter. You can check the filter by looking at its frequency response.

References

[1] Parks, T.W., and C.S. Burrus, *Digital Filter Design*, John Wiley & Sons, 1987, pp. 54-83.

[2] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, 1989, pp. 256-266.

See Also   fir1 | fir2 | firrcos | firpm

Purpose	Minimum-phase FIR spectral factor
Syntax	`h = firminphase(b)` `h = firminphase(b,nz)`

Description

`h = firminphase(b)` computes the minimum-phase FIR spectral factor `h` of a linear-phase FIR filter `b`. Filter `b` must be real, have even order, and have nonnegative zero-phase response.

`h = firminphase(b,nz)` specifies the number of zeros, `nz`, of `b` that lie on the unit circle. You must specify `nz` as an even number to compute the minimum-phase spectral factor because every root on the unit circle must have even multiplicity. Including `nz` can help `firminphase` calculate the required FIR spectral factor. Zeros with multiplicity greater than two on the unit circle cause problems in the spectral factor determination.

Note You can find the maximum-phase spectral factor, g, by reversing h, such that $g = fliplr(h)$, and $b = conv(h, g)$.

Examples

This example designs a constrained least squares filter with a nonnegative zero-phase response, and then uses `firminphase` to compute the minimum-phase spectral factor.

```
f  = [0 0.4 0.8 1];
a  = [0 1 0];
up = [0.02 1.02  0.01];
lo = [0 0.98 0]; % The zeros insure nonnegative zero-phase resp.
n  = 32;
b  = fircls(n,f,a,up,lo);
h  = firminphase(b);
```

References

Saramaki, T, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

firminphase

See Also   firgr | fircls | zerophase

Purpose   Lowpass Nyquist (Lth-band) FIR filter

Syntax
```
b = firnyquist(n,l,r)
b = firnyquist('minorder',l,r,dev)
b = firnyquist(n,l,r,decay)
b = firnyquist(n,l,r,'nonnegative')
b = firnyquist(n,l,r,'minphase')
```

Description   `b = firnyquist(n,l,r)` designs an Nth order, Lth band, Nyquist FIR filter with a rolloff factor `r` and an equiripple characteristic.

The rolloff factor `r` is related to the normalized transition width `tw` by $tw = 2\pi(r/l)$ (rad/sample). The order, n, must be even. `l` must be an integer greater than one. If `l` is not specified, it defaults to `4`. `r` must satisfy `0< r < 1`. If `r` is not specified, it defaults to `0.5`.

`b = firnyquist('minorder',l,r,dev)` designs a minimum-order, Lth band Nyquist FIR filter with a rolloff factor `r` using the Kaiser window. The peak ripple is constrained by the scalar `dev`.

`b = firnyquist(n,l,r,decay)` designs an Nth order (n), Lth band (l), Nyquist FIR filter where the scalar `decay`, specifies the rate of decay in the stopband. `decay` must be nonnegative. If you omit or leave it empty, `decay` defaults to `0` which yields an equiripple stopband. A nonequiripple stopband (decay $\neq 0$) may be desirable for decimation purposes.

`b = firnyquist(n,l,r,'nonnegative')` returns an FIR filter with nonnegative zero-phase response. This filter can be spectrally factored into minimum-phase and maximum-phase "square-root" filters. This allows you to use the spectral factors in applications such as matched-filtering.

`b = firnyquist(n,l,r,'minphase')` returns the minimum-phase spectral factor `bmin` of order n. `bmin` meets the condition `b=conv(bmin,bmax)` so that b is an Lth band FIR Nyquist filter of order 2n with filter rolloff factor `r`. Obtain `bmax`, the maximum phase spectral factor by reversing the coefficients of `bmin`. For example, `bmax = bmin(end:-1:1)`.

firnyquist

Examples   **Example 1**

This example designs a minimum phase factor of a Nyquist filter.

```
bmin = firnyquist(47,10,.45,'minphase');
b = firnyquist(2*47,10,.45,'nonnegative');
[h,w,s] = freqz(b); hmin = freqz(bmin);
fvtool(b,1,bmin,1);
```

Example 2

This example compares filters with different decay rates.

```
b1 = firnyquist(72,8,.3,0); % Equiripple
b2 = firnyquist(72,8,.3,15);
b3 = firnyquist(72,8,.3,25);
fvtool(b1,1,b2,1,b3,1);
```

References   T. Saramaki, Finite Impulse Response Filter Design, *Handbook for Digital Signal Processing*, Mitra, S.K. and J.F. Kaiser Eds. Wiley-Interscience, N.Y., 1993, Chapter 4.

See Also   firhalfband | firgr | firls | firminphase | firrcos | firls

Purpose   Two-channel FIR filter bank for perfect reconstruction

Syntax
```
[h0,h1,g0,g1] = firpr2chfb(n,fp)
[h0,h1,g0,g1] = firpr2chfb(n,dev,'dev')
[h0,h1,g0,g1] = firpr2chfb('minorder',fp,dev)
```

Description   `[h0,h1,g0,g1] = firpr2chfb(n,fp)` designs four FIR filters for the analysis sections (`h0` and `h1`) and synthesis section is (`g0` and `g1`) of a two-channel perfect reconstruction filter bank. The design corresponds to the orthogonal filter banks also known as power-symmetric filter banks.

`n` is the order of all four filters. It must be an odd integer. `fp` is the passband-edge for the lowpass filters `h0` and `g0`. The passband-edge argument `fp` must be less than 0.5. `h1` and `g1` are highpass filters with the passband-edge given by (1-`fp`).

`[h0,h1,g0,g1] = firpr2chfb(n,dev,'dev')` designs the four filters such that the maximum stopband ripple of `h0` is given by the scalar `dev`. Specify `dev` in linear units, not decibels. The stopband-ripple of `h1` is also be given by `dev`, while the maximum stopband-ripple for both `g0` and `g1` is (2*`dev`).

`[h0,h1,g0,g1] = firpr2chfb('minorder',fp,dev)` designs the four filters such that `h0` meets the passband-edge specification `fp` and the stopband-ripple `dev` using minimum order filters to meet the specification.

Examples   Design a filter bank with filters of order `n` equal to 99 and passband edges of 0.45 and 0.55.

```
n = 99;
[h0,h1,g0,g1] = firpr2chfb(n,.45);
fvtool(h0,1,h1,1,g0,1,g1,1);
```

Here are the filters, showing clearly the passband edges.
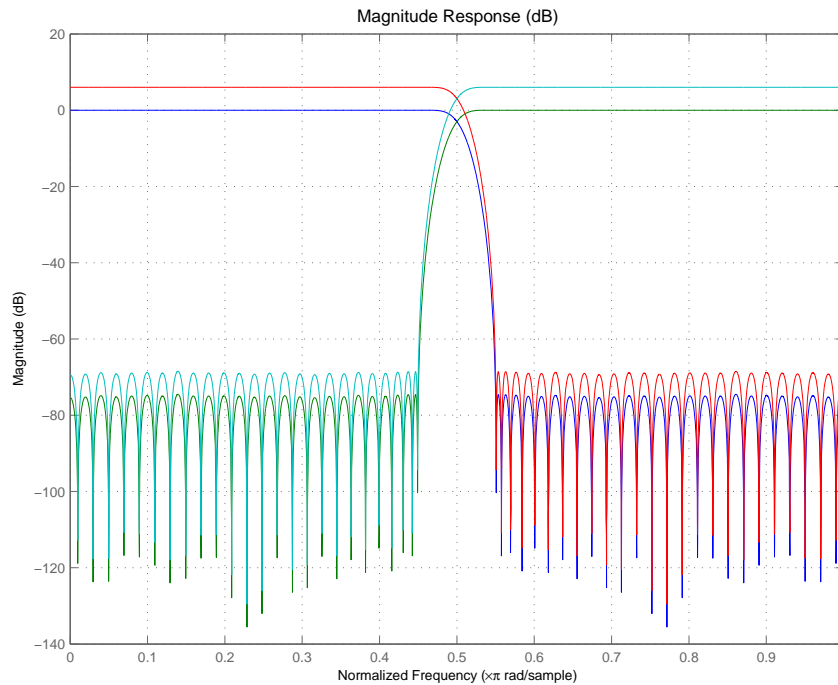
Use the following stem plots to verify perfect reconstruction using the
filter bank created by `firpr2chfb`.

```
stem(1/2*conv(g0,h0)+1/2*conv(g1,h1))
n=0:n;
stem(1/2*conv((-1).^n.*h0,g0)+1/2*conv((-1).^n.*h1,g1))
stem(1/2*conv((-1).^n.*g0,h0)+1/2*conv((-1).^n.*g1,h1))
stem(1/2*conv((-1).^n.*g0,(-1).^n.*h0)+...
1/2*conv((-1).^n.*g1,(-1).^n.*h1))
stem(conv((-1).^n.*h1,h0)-conv((-1).^n.*h0,h1))
```

See Also   `firceqrip | firgr | firhalfband | firnyquist`

Purpose   Type of linear phase FIR filter

Syntax   t = firtype(hd)
               t = firtype(hm)
               t = firtype(hs)

Description t = firtype(hd) determines the type (1 through 4) of a discrete-time
                FIR filter object hd, returning the type number in t. Filter hd must be
                real and have linear phase.

                t = firtype(hm) determines the type (1 through 4) of the multirate
                filter object hm. The filter must be real and have linear phase. When hm
                has multiple sections, all sections must be real FIR filters with linear
                phase. In this case, t is a cell array containing the filter type of each
                section.

                t = firtype(hs) determines the type (1 through 4) of the filter System
                object hs. The filter must be real and have linear phase.

Input   **hd**
Arguments
                Discrete-time dfilt filter object.

                hm

                Multirate mfilt filter object.

                hs

                Filter System object. The function supports the following filter
                structures.

                - dsp.FIRFilter

                - dsp.BiquadFilter

                - dsp.FIRInterpolator

                - dsp.CICInterpolator

                - dsp.FIRDecimator

firtype

- dsp.CICDecimator
- dsp.FIRRateConverter

Output Arguments

t

Filter type. t is either 1, 2, 3, or 4. These types are defined as follows:

- Type 1 — Even-order symmetric coefficients
- Type 2 — Odd-order symmetric coefficients
- Type 3 — Even-order antisymmetric coefficients
- Type 4 — Odd-order antisymmetric coefficients

Examples

Determine the type of the default interpolator for L=4.

```
l = 4;
hm = mfilt.firinterp(l);
firtype(hm)
ans =

     1
```

See Also

islinphase

Purpose   Frequency response estimate via filtering

Syntax   ```
[h,w] = freqrespest(H,L)
[h,w] = freqrespest(H,L,param1,value1,param2,value2,...)
[h,w] = freqrespest(H,L,opts)
freqrespest(H,...)
```

**Description**      `[h,w] = freqrespest(H,L)` estimates the frequency response of a
`dfilt` object or a filter System object. A set of input data is filtered
and then forming the ratio between output data and input data. The
test input data comprises sinusoids with uniformly distributed random
frequencies.

Use this technique for comparing the performance of fixed-point
filters to that of another filter type. You can, for example, compare
fixed—point frequency response estimate to that of a similar filter
that uses quantized coefficients, but applies floating-point arithmetic
internally. Such comparison determines whether the fixed-point filter
performance closely matches the floating-point, quantized coefficients
version of the filter.

`L` is the number of trials to use to compute the estimate. If you do
not specify this value, `L` defaults to 10. More trials generates a more
accurate estimate of the response, but require more time to compute
the estimate.

`h` is the estimate of the complex frequency response. `w` contains the
vector of frequencies at which `h` is estimated.

`[h,w] = freqrespest(H,L,param1,value1,param2,value2,...)`
accepts H as either a `dfilt` object or a filter System object. This
approach uses parameter value (PV) pairs as input arguments to specify
optional parameters for the test. These parameters are the valid PV
pairs. Enter the parameter names as string input arguments in single
quotation marks. The following table provides valid parameters for
`[h, w]`.

| Parameter Name | Default Value | Description |
|---|---|---|
| NFFT | 512 | Number of FFT points to use. |
| NormalizedFrequency | true | Indicate whether to use normalized frequency or linear frequency. Values are true (use normalized frequency), or false (use linear frequency). When you specify false, you must supply the sampling frequency Fs. |
| Fs | normalized | Specify the sampling frequency when NormalizedFrequency is false. No default value. You must set NormalizedFrequency to false before setting a value for Fs. |
| SpectrumRange | half | Specify the spectrum range to be used as whole or half (default). |
| CenterDC | false | Specify whether to set the center of the spectrum to the DC value in the output plot. If you select true, both the negative and positive values appear in the plot. If you select false, DC appears at the origin of the axes. |
| Arithmetic (only for filter System objects) | ARITH | Analyze the filter System object, based on the arithmetic specified in the ARITH input. ARITH can be set to double, single, or fixed. The analysis tool assumes a double-precision |

| Parameter Name | Default Value | Description |
|---|---|---|
| | | filter when the arithmetic input is not specified and the filter System object is in an unlocked state. |

If H is a filter System object, freqrespest requires knowledge of the input data type. Analysis cannot be performed if the input data type is not available. If you do not specify the Arithmetic parameter, i.e., use the syntax [h,w] = freqrespest(H) , then the following rules apply for this method:

- The System object state is Unlocked — freqrespest performs double precision analysis.

- The System object state is Locked — freqrespest performs analysis based on the locked input data type.

When you do specify the Arithmetic parameter, i.e., use the syntax [h,w] = freqrespest(H,'Arithmetic', ARITH)), review the following rules for this method. Which rule applies depends on the value you set for the Arithmetic parameter.

| Value | System Object State | Rule |
|---|---|---|
| ARITH = 'double' | Unlocked | freqrespest performs double-precision analysis. |
| | Locked | freqrespest performs double-precision analysis. |

| Value | System Object State | Rule |
|---|---|---|
| ARITH = `single' | Unlocked | freqrespest performs single-precision analysis. |
| | Locked | freqrespest performs single-precision analysis. |
| ARITH = `fixed' | Unlocked | freqrespest produces an error because the fixed-point input data type is unknown. |
| | Locked | If the input data type is double or single, then freqrespest produces an error because the fixed-point input data type is unknown. |
| | | When the input data is of fixed-point type, freqrespest performs analysis based on the locked input data type. |

freqrespest supports only the following filter System objects:

- dsp.FIRFilter
- dsp.IIRFilter
- dsp.AllpoleFilter

• `dsp.BiquadFilter`

Regardless of whether H is a `dfilt` object or a filter System object, `[h,w] = freqrespest(H,L,opts)` uses an object, `opts`, to specify the optional input parameters. This specification is not done directly by specifying PV pairs as input arguments. Create `opts` with

```
opts = freqrespopts(H);
```

Because `opts` is an object, you use `set` to change the parameter values in `opts` before you use it with `freqrespest`. For example, you could specify a new sample rate with

```
set(opts,'fs',48e3); % Same as opts.fs=48e3
```

Regardless of whether H is a `dfilt` object or a filter System object, `freqrespest(H,...)` with no output argument launches FVTool.

`freqrespest` can also compute the frequency response of double-precision floating filters. Such filters cannot be converted to transfer-function form without introducing significant round off errors which affect the `freqz` frequency response computation. Examples of these kinds of filters include state-space or lattice filters, especially if they are high-order filters.

Examples
These examples demonstrate some uses for `freqrespest`.

Example 1

Start by estimating the frequency response of a fixed-point FIR filter that has filter internals set to `full precision`.

```
hd = design(fdesign.lowpass(.4,.5,1,60),'equiripple');
hd.arithmetic = 'fixed';
[h,w] = freqrespest(hd); % This should be about the same as freqz.
```

Continuing with filter hd, change the value of the `filterinternals` property to `specifyprecision` and then specify the word lengths and precision (the fraction lengths) applied to the output from internal addition and multiplication operations. After you set the word and

fraction lengths, use `freqrespest` to compute the frequency response estimate for the fixed-point filter.

```
hd.filterinternals = 'specifyprecision';
hd.outputwordlength=16;
hd.outputfraclength=15;
hd.productwordlength=16;
hd.productfraclength=15;
hd.accumwordlength=16;
hd.accumfraclength=15;
[h,w] = freqrespest(hd,2);
[h2,w2] = freqz(hd,512);
plot(w/pi,20*log10(abs([h,h2])))
legend('Frequency response estimated by filtering',...
'Freq. response computed by quantizing coefficients only');
xlabel('Normalized Frequency (\times\pi rad/sample)')
ylabel('Magnitude (dB)')
```

Example 2

freqrespest works with state-space filters as well. This example estimates the frequency response of a state-space filter.

```
fs = 315000;
wp = [320 3800]/(fs/2);
ws = [50 19000]/(fs/2);
rp=0.15; rs=60;
[n,wn]=cheb1ord(wp,ws,rp,rs);
[a,b,c,d] = cheby1(n,rp,wn);
hd = dfilt.statespace(a,b,c,d);
% Compare the following to freqz(hd,8192)
freqrespest(hd,1,'nfft',8192);
```

freqrespest

See Also   dfilt | freqrespopts | freqz | limitcycle | noisepsd | scale

Purpose   Options for filter frequency response analysis

Syntax   opts = freqrespopts(H)

Description   opts = freqrespopts(H) uses the settings in the dfilt object or the
filter System object, H, to create an object, opts. This object contains
parameters and values for estimating the filter frequency response.
You pass opts as an input argument to freqrespest to specify values
for the input parameters.

freqrespopts allows you to use the same settings for freqrespest
with multiple filters without specifying all of the parameters as input
arguments to freqrespest.

freqrespopts supports only the following filter System objects:

- dsp.FIRFilter
- dsp.IIRFilter
- dsp.AllpoleFilter
- dsp.BiquadFilter

Examples   This example shows freqrespopts in use for setting options for
freqrespest. hd and hd2 are bandpass filters that use different design
methods. The opts object makes it easier to set the same conditions for
the frequency response estimate in freqrespest.

```
d=fdesign.bandpass('fst1,fp1,fp2,fst2,ast1,ap,ast2',...
0.25,0.3,0.45,0.5,60,0.1,60);

hd=design(d,'butter');
hd.arithmetic='fixed';
hd2=design(d,'cheby2')
hd2.arithmetic='fixed';
opts=freqrespopts(hd)

opts =
```

```
                         NFFT: 512
        NormalizedFrequency: true
                           Fs: 'Normalized'
               SpectrumRange: 'Half'
                     CenterDC: false

opts.NFFT=256; % Same as set(opts,'nfft',256).
opts.NormalizedFrequency=false;
opts.fs=1.5e3;
opts.CenterDC=true

opts =

                         NFFT: 256
        NormalizedFrequency: false
                           Fs: 1500
               SpectrumRange: 'Whole'
                     CenterDC: true
```

With opts configured as needed, use it as an input argument for
freqrespest.

```
[h2,w2]=freqrespest(hd2,20,opts);
[h1,w1]=freqrespest(hd,20,opts);
```

See Also freqrespest | noisepsd | noisepsdopts | norm | scale

Purpose   Real or complex frequency-sampled FIR filter from specification object

Syntax   hd = design(d,'freqsamp')
                   hd = design(...,'filterstructure',structure)
                   hd = design(...,'window',window)

Description   hd = design(d,'freqsamp') designs a frequency-sampled filter
                   specified by the filter specifications object d.

                   hd = design(...,'filterstructure',structure) returns a filter with
                   the filter structure you specify by the structure input argument.
                   structure is dffir by default and can be any one of the following filter
                   structures.

Structure String	Description of Resulting Filter Structure
dffir	Direct-form FIR filter
dffirt	Transposed direct-form FIR filter
dfsymfir	Symmetrical direct-form FIR filter
dfasymfir	Asymmetrical direct-form FIR filter
fftfir	Fast Fourier transform FIR filter

hd = design(...,'window',window) designs filters using the window
specified by the string in window. Provide the input argument window as

- A string for the window type. For example, use 'bartlett', or
  'hamming'. See window for the full list of windows available or refer
  to window in the *Signal Processing Toolbox User's Guide*.

- A function handle that references the window function. When the
  window function requires more than one input, use a cell array to
  hold the required arguments. The first example shows a cell array
  input argument.

- The window vector itself.

Examples   These examples design FIR filters that have arbitrary magnitude responses. In the first filter, the response has three distinct sections and the resulting filter is real.

The second example creates a complex filter.

```
b1 = 0:0.01:0.18;
b2 = [.2 .38 .4 .55 .562 .585 .6 .78];
b3 = [0.79:0.01:1];
a1 = .5+sin(2*pi*7.5*b1)/4;    % Sinusoidal response section.
a2 = [.5 2.3 1 1 -.2 -.2 1 1]; % Piecewise linear response section.
a3 = .2+18*(1-b3).^2;          % Quadratic response section.
f = [b1 b2 b3];
a = [a1 a2 a3];
n = 300;
d = fdesign.arbmag('n,f,a',n,f,a); % First specifications object.
hd = design(d,'freqsamp','window',{@kaiser,.5}); % Filter.
fvtool(hd)
```

The plot from FVTool shows the response for hd.

Magnitude Response (dB)

Now design the arbitrary-magnitude complex FIR filter. Recall that vector f contains frequency locations and vector a contains the desired filter response values at the locations specified in f.

```
f = [-1 -.93443 -.86885 -.80328 -.7377 -.67213 -.60656 -.54098 ...
-.47541,-.40984 -.34426 -.27869 -.21311 -.14754 -.081967 ...
-.016393 .04918 .11475,.18033 .2459 .31148 .37705 .44262 ...
 .5082 .57377 .63934 .70492 .77049,.83607 .90164 1];
a = [.0095848 .021972 .047249 .099869 .23119 .57569 .94032 ...
.98084 .99707,.99565 .9958 .99899 .99402 .99978 .99995 .99733 ...
.99731 .96979 .94936,.8196 .28502 .065469 .0044517 .018164 ...
.023305 .02397 .023141 .021341,.019364 .017379 .016061];
n = 48;
d = fdesign.arbmag('n,f,a',n,f,a); % Second spec. object.
hdc = design(d,'freqsamp','window','rectwin'); % Filter.
fvtool(hdc)
```

FVTool shows you the response for `hdc` from -1 to 1 in normalized frequency. `design(d,...)` returns a complex filter for `hdc` because the frequency vector includes negative frequency values.

Magnitude Response (dB)

See Also   `design` | `designmethods` | `fdesign.arbmag` | `help` | `window`

Purpose   Frequency response of filter

Syntax
```
[h,w] = freqz(hfilt)
[h,w] = freqz(hfilt,n)
freqz(hfilt)
[h,w] = freqz(hs)
[h,w] = freqz(hs,n)
[h,w] = freqz(hs,Name,Value)
freqz(hs)
```

Description   freqz returns the frequency response based on the current filter
coefficients. This section describes common freqz operation with
adaptive filters, discrete-time filters, multirate filters, and filter System
objects. For more input options, refer to freqz in Signal Processing
Toolbox documentation.

[h,w] = freqz(hfilt) returns the frequency response h and the
corresponding frequencies w at which the filter response of hfilt is
computed. The frequency response is evaluated at 8192 points equally
spaced around the upper half of the unit circle.

[h,w] = freqz(hfilt,n) returns the frequency response h and
corresponding frequencies w for the filter or vector of filters hfilt. The
frequency response is evaluated at n points equally spaced around the
upper half of the unit circle. freqz uses the transfer function associated
with the filter to calculate the frequency response of the filter with the
current coefficient values.

freqz(hfilt) uses FVTool to plot the magnitude and unwrapped phase
of the frequency response of the filter hfilt. If hfilt is a vector of
filters, freqz plots the magnitude response and phase for each filter in
the vector.

[h,w] = freqz(hs) returns a frequency response for the filter System
object hs using 8192 samples.

[h,w] = freqz(hs,n) returns a frequency response for the filter
System object hs using n samples.

freqz

[h,w] = freqz(hs,Name,Value) returns a frequency response
with additional options specified by one or more Name,Value pair
arguments.

freqz(hs) uses FVTool to plot the magnitude and unwrapped phase of
the frequency response of the filter System object hs.

Input Arguments

hfilt

hfilt is either:

- An adaptive adaptfilt, discrete-time dfilt, or multirate mfilt
  filter object

- A vector of adaptive, discrete-time, or multirate filter objects

hs

Filter System object. The function supports the following filter
structures.

- dsp.FIRFilter

- dsp.BiquadFilter

- dsp.FIRInterpolator

- dsp.CICInterpolator

- dsp.FIRDecimator

- dsp.CICDecimator

- dsp.FIRRateConverter

n

Number of samples. For an FIR filter where n is a power of two, the
computation is done faster using FFTs.

  Default: 8192

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Arithmetic
`double' | 'single' | 'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

System Object State	Coefficient Data Type	Rule
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

h

Complex, n-element frequency response vector. If hfilt is a vector of filters, h is a complex, length(hfilt)-by-n matrix of frequency response vectors corresponding to each filter in hfilt. If n is not specified, the function uses a default value of 8192.

For adaptive filters, h is the instantaneous frequency response.

w

Frequency vector of length n, in radians/sample. w consists of n points equally spaced around the upper half of the unit circle (from 0 to π radians/sample). If n is not specified, the function uses a default value of 8192.

Tips   There are several ways of analyzing the frequency response of filters. freqz accounts for quantization effects in the filter coefficients, but does not account for quantization effects in filtering arithmetic. To account for the quantization effects in filtering arithmetic, refer to function noisepsd.

Algorithms   freqz calculates the frequency response for a filter from the filter transfer function $Hq(z)$. The complex-valued frequency response is calculated by evaluating $Hq(e^{j\omega})$ at discrete values of w specified by the syntax you use. The integer input argument n determines the number of equally-spaced points around the upper half of the unit circle at which freqz evaluates the frequency response. The frequency ranges from 0 to π radians per sample when you do not supply a sampling frequency as an input argument. When you supply the scalar sampling frequency fs as an input argument to freqz, the frequency ranges from 0 to fs/2 Hz.

Examples   Plot the estimated frequency response of a filter. This example uses discrete-time filters, but hd can be any adaptfilt object, dfilt object, mfilt object, or filter System object. First plot the results for one filter.

```
b = fir1(80,0.5,kaiser(81,8));
hd = dfilt.dffir(b);
freqz(hd);
```

If you have more than one filter, you can plot them on the same figure using a vector of filters.

```
b = fir1(40,0.5,kaiser(41,6));
hd2 = dfilt.dffir(b);
h = [hd hd2];
freqz(h);
```

See Also   adaptfilt | dfilt | mfilt | fvtool

gain

Purpose	CIC filter gain

Syntax

```
gain(hm)
gain(hm,j)
gain(hs)
```

Description   gain(hm) returns the gain of hm, the CIC decimation or interpolation filter.

When hm is a decimator, gain returns the gain for the overall CIC decimator.

When hm is an interpolator, the CIC interpolator inserts zeros into the input data stream, reducing the filter overall gain by $1/R$, where R is the interpolation factor, to account for the added zero valued samples. Therefore, the gain of a CIC interpolator is $(RM)^N/R$, where N is the number of filter sections and M is the filter differential delay. gain(hm) returns this value. The next example presents this case.

gain(hm,j) returns the gain of the jth section of a CIC interpolation filter. When you omit j, gain assumes that j is 2*N, where N is the number of sections, and returns the gain of the last section of the filter. This syntax does not apply when hm is a decimator.

gain(hs) returns the gain of the filter System object hs. The function supports the dsp.CICDecimator and dsp.CICInterpolator filter structures.

Examples   To compare the performance of two interpolators, one a CIC filter and the other an FIR filter, use gain to adjust the CIC filter output amplitude to match the FIR filter output amplitude. Start by creating an input data set—a sinusoidal signal x.

```
fs = 1000;          % Input sampling frequency.
t = 0:1/fs:1.5;     % Signal length = 1501 samples.
x = sin(2*pi*10*t); % Amplitude = 1 sinusoid.

l = 4; % Interpolation factor for FIR filter.
d = fdesign.interpolator(l);
```

```
hm = design(d,'multistage');
ym = filter(hm,x);

r = 4; % Interpolation factor for the CIC filter.
d = fdesign.interpolator(r,'cic');
hcic = design(d,'multisection');
ycic = filter(hcic,x);
gaincic = gain(hcic);
subplot(211);
plot(1:length(ym),[ym; double(ycic)]);
subplot(212)
plot(1:length(ym),[ym; double(ycic)/gain(hcic)]);
```

After correcting for the gain induced by the CIC interpolator, the figure shows the filters provide nearly identical interpolation.

See Also   scale

grpdelay

Purpose   Filter group delay

Syntax
```
[gd,w]=grpdelay(hfilt)
[gd,w]=grpdelay(hfilt,n)
grpdelay(hfilt)
[gd,w] = grpdelay(hs)
[gd,w] = grpdelay(hs,n)
[gd,w] = grpdelay(hs,Name,Value)
grpdelay(hs)
```

Description   grpdelay returns the group delay based on the current filter coefficients. This section describes common grpdelay operation with adaptive filters, discrete-time filters, multirate filters, and filter System objects. For more input options, refer to grpdelay in Signal Processing Toolbox documentation.

[gd,w]=grpdelay(hfilt) returns the phase response phi of the filter hfilt and the corresponding frequencies w at which the function evaluates the group delay. Group delay is

$$-\frac{d}{dw}(angle(w))$$

The group delay is evaluated at 8192 points equally spaced around the upper half of the unit circle.

[gd,w]=grpdelay(hfilt,n) returns the group delay phi of the filter hfilt and the corresponding frequencies w at which the function evaluates the group delay. The group delay is evaluated at n points equally spaced around the upper half of the unit circle.

grpdelay(hfilt) displays the group delay of hfilt in the Filter Visualization Tool (FVTool).

[gd,w] = grpdelay(hs) returns the group delay for the filter System object hs using 8192 samples.

[gd,w] = grpdelay(hs,n) returns the group delay for the filter System object hs using n samples.

[gd,w] = grpdelay(hs,Name,Value) returns the group delay with additional options specified by one or more Name,Value pair arguments.

grpdelay(hs) uses FVTool to plot the group delay of the filter System object hs.

Input Arguments

hfilt

hfilt is either:

- An adaptive adaptfilt, discrete-time dfilt, or multirate mfilt filter object

- A vector of adaptive, discrete-time, or multirate filter objects

hs

Filter System object. The function supports the following filter structures.

- dsp.FIRFilter

- dsp.BiquadFilter

- dsp.FIRInterpolator

- dsp.CICInterpolator

- dsp.FIRDecimator

- dsp.CICDecimator

- dsp.FIRRateConverter

n

Number of samples. For an FIR filter where n is a power of two, the computation is done faster using FFTs.

Default: 8192

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Arithmetic

`double' | 'single' | 'fixed'

For filter System object inputs only, specify the arithmetic used during analysis. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed', the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

System Object State	Coefficient Data Type	Rule
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

gd

Complex, n-element group-delay vector. If hfilt is a vector of filters, gd is a complex, length(hfilt)-by-n matrix of group-delay vectors whose columns correspond to each filter in hfilt. If n is not specified, the function uses a default value of 8192.

For adaptive filters, gd is the instantaneous group delay.

w

Frequency vector of length n, in radians/sample. w consists of n points equally spaced around the upper half of the unit circle (from 0 to π radians/sample). If hfilt is a vector of filters, w is a complex, length(hfilt)-by-n matrix of group-delay vectors whose columns correspond to each filter in hfilt. If n is not specified, the function uses a default value of 8192.

See Also   phasez | zerophase

Purpose   Help for design method with filter specification

Syntax   help(d,'*designmethod*')

Description help(d,'*designmethod*') displays help in the Command Window for
                the design algorithm designmethod for the current specifications of the
                filter specification object d. The string you enter for designmethod must
                be one of the strings returned by designmethods for d, the design object.

Examples   Get specific help for designing lowpass Butterworth filters. The first
                lowpass filter uses the default specification string 'Fp,Fst,Ap,Ast' and
                returns help text specific to the specification string.

```
d = fdesign.lowpass;
designmethods(d)

Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):

butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage

help(d,'butter')

DESIGN Design a Butterworth IIR filter.
HD = DESIGN(D, 'butter') designs a Butterworth filter specified
by the FDESIGN object D.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a filter
with the structure STRUCTURE.  STRUCTURE is 'df2sos' by default
and can be any of the following.
```

```
'df1sos'
'df2sos'
'df1tsos'
'df2tsos'

HD = DESIGN(..., 'MatchExactly', MATCH) designs a Butterworth
filter and matches the frequency and magnitude specification for
the band MATCH exactly. The other band will exceed the
specification.  MATCH can be 'stopband' or 'passband' and is
'stopband' by default.

% Example #1 - Compare passband and stopband MatchExactly.
h    = fdesign.lowpass('Fp,Fst,Ap,Ast', .1, .3, 1, 60);
Hd   = design(h, 'butter', 'MatchExactly', 'passband');
Hd(2) = design(h, 'butter', 'MatchExactly', 'stopband');

% Compare the passband edges in FVTool.
fvtool(Hd);
axis([.09 .11 -2 0]);
```

Note the discussion of the `MatchExactly` input option. When you use a design object that uses a different specification string, such as 'N,F3dB', the help content for the `butter` design method changes.

In this case, the `MatchExactly` option does not appear in the help because it is not an available input argument for the specification string 'N,F3dB'.

```
d=fdesign.lowpass('N,F3dB')

d =
                 Response: 'Lowpass'
            Specification: 'N,F3dB'
              Description: {'Filter Order';'3dB Frequency'}
      NormalizedFrequency: true
              FilterOrder: 10
                     F3dB: 0.5
```

```
designmethods(d)

Design Methods for class fdesign.lowpass (N,F3dB):

butter

help(d,'butter

DESIGN Design a Butterworth IIR filter.

HD = DESIGN(D, 'butter') designs a Butterworth filter
specified by the FDESIGN object D.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns a
filter with the structure STRUCTURE. STRUCTURE is 'df2sos'
by default and can be any of the following.

'df1sos'

'df2sos'

'df1tsos'

'df2tsos'

% Example #1 - Design a lowpass Butterworth filter in the
DF2TSOS structure.

h = fdesign.lowpass('N,F3dB');

Hd = design(h, 'butter', 'FilterStructure', 'df2tsos');
```

See Also   fdesign | design | designmethods | designopts

ifir

Purpose	Interpolated FIR filter from filter specification

Syntax	`hd = ifir(d)`
	`hd = design(d,'ifir',designoption,value,designoption,...`
	`value,...)`

Description   `hd = ifir(d)` designs an FIR filter from design object `d`, using the interpolated FIR method. `ifir` returns `hd` as a cascade of two filters that act together to meet the specifications in `d`. The resulting filter is particularly efficient, having a low number of multipliers. However, if `ifir` determines that a single-stage filter would be more efficient than the default two-stage design, it returns `hd` as a single-stage filter. `ifir` only creates linear phase filters. Generally, `ifir` uses an advanced optimization algorithm to create highly efficient FIR filters.

`ifir` returns `hd` as either a single-rate `dfilt` object or a multirate `mfilt` object (when you have DSP System Toolbox software installed), based on the specifications you provide in `d`, the filter specification object.

specifications supplied in the object `h`.

`hd = design(d,'ifir',designoption,value,designoption,...`
`value,...)` returns an interpolated FIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

`designopts(d,'method')`

For complete help about using `ifir`, refer to the command line help system. For example, to get specific information about using `ifir` with `d`, the specification object, enter the following at the MATLAB prompt.

`help(d,'ifir')`

Note For help about how you use `ifir` to design filters without using design objects, enter

```
help ifir
```

at the MATLAB prompt.

Examples   Use `fdesign.lowpass` and `fdesign.highpass` to design a lowpass filter and a wideband highpass filter. After designing the filters, use FVTool to plot the response curves for both.

```
fpass = 0.2;
fstop = 0.24;
d1 = fdesign.lowpass(fpass, fstop);
hd1 = design(d1,'ifir');
fstop = 0.2;
fpass = 0.25;
astop = 40;
apass = 1;
d2 = fdesign.highpass(fstop,fpass,astop,apass);
hd2 = design(d2,'ifir');
fvtool(hd1,hd2)
```

See Also   fdesign | firgr | fir1 | firls | firpm

Purpose   Transform IIR complex bandpass filter to IIR complex bandpass filter
                     with different characteristics

Syntax   [Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)

Description   [Num,Den,AllpassNum,AllpassDen] = iirbpc2bpc(B,A,Wo,Wt)
                     returns the numerator and denominator vectors, Num and Den
                     respectively, of the target filter transformed from the complex bandpass
                     prototype by applying a first-order complex bandpass to complex
                     bandpass frequency transformation.

                     It also returns the numerator, AllpassNum, and the denominator,
                     AllpassDen, of the allpass mapping filter. The prototype lowpass
                     filter is given with the numerator specified by B and the denominator
                     specified by A.

                     This transformation effectively places two features of an original
                     filter, located at frequencies W_{o1} and W_{o2}, at the required target
                     frequency locations, W_{t1}, and W_{t2} respectively. It is assumed that W_{t2}
                     is greater than W_{t1}. In most of the cases the features selected for the
                     transformation are the band edges of the filter passbands. In general
                     it is possible to select any feature; e.g., the stopband edge, the DC, the
                     deep minimum in the stopband, or other ones.

                     Relative positions of other features of an original filter do not change in
                     the target filter. This means that it is possible to select two features of
                     an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
                     precede F_2 after the transformation. However, the distance between F_1
                     and F_2 will not be the same before and after the transformation.

                     This transformation can also be used for transforming other types of
                     filters; e.g., complex notch filters or resonators can be repositioned at
                     two distinct desired frequencies at any place around the unit circle;
                     e.g., in the adaptive system.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
                     approach:

                     [b, a] = ellip(3, 0.1, 30, 0.409);

Create a complex passband from 0.25 to 0.75:

```
[b, a] = iirlp2bpc (b, a, 0.5, [0.25,0.75]);
[num, den] = iirbpc2bpc(b, a, [0.25, 0.75], [-0.5, 0.5]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Using FVTool to plot the filters shows you the comparison, presented in this figure.

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also   iirftransf | allpassbpc2bpc | zpkbpc2bpc

iircomb

Purpose   IIR comb notch or peak filter

Syntax
```
[num,den] = iircomb(n,bw)
[num,den] = iircomb(n,bw,ab)
[num,den] = iircomb(...,'type')
```

Description   `[num,den] = iircomb(n,bw)` returns a digital notching filter with order `n` and with the width of the filter notch at -3 dB set to `bw`, the filter bandwidth. The filter order must be a positive integer. `n` also defines the number of notches in the filter across the frequency range from 0 to 2π — the number of notches equals `n+1`.

For the notching filter, the transfer function takes the form

$$H(z) = b\frac{1 - z^{-n}}{1 - \alpha z^{-n}}$$

where α and b are the positive scalars and n is the filter order or the number of notches in the filter minus 1.

The quality factor (Q factor) `q` for the filter is related to the filter bandwidth by $q = \omega_0/bw$ where ω_0 is the frequency to remove from the signal.

`[num,den] = iircomb(n,bw,ab)` returns a digital notching filter whose bandwidth, `bw`, is specified at a level of `-ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default -3 dB point, such as -6 dB or 0 dB.

`[num,den] = iircomb(...,'type')` returns a digital filter of the specified type. The input argument `type` can be either

- `'notch'` to design an IIR notch filter. Notch filters attenuate the response at the specified frequencies. This is the default type. When you omit the `type` input argument, `iircomb` returns a notch filter.

- `'peak'` to design an IIR peaking filter. Peaking filters boost the signal at the specified frequencies.

The transfer function for peaking filters is

$$H(z) = b \frac{1 - z^{-n}}{1 + az^{-n}}$$

Examples   Design and plot an IIR notch filter with 11 notches (equal to filter order plus 1) that removes a 60 Hz tone (f0) from a signal at 600 Hz (fs). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth.

```
fs = 600; fo = 60;  q = 35; bw = (fo/(fs/2))/q;
[b,a] = iircomb(fs/fo,bw,'notch'); % Note type flag 'notch'
fvtool(b,a);
```

Using the Filter Visualization Tool (FVTool) generates the following plot showing the filter notches. Note the notches are evenly spaced and one falls at exactly 60 Hz.

See Also   firgr | iirnotch | iirpeak

Purpose   IIR frequency transformation of filter

Syntax   `[OutNum,OutDen] = iirftransf(OrigNum,OrigDen,FTFNum,FTFDen)`

Description   `[OutNum,OutDen] = iirftransf(OrigNum,OrigDen,FTFNum,FTFDen)`
returns the numerator and denominator vectors, `OutNum` and `OutDen`,
of the target filter, which is the result of transforming the prototype
filter specified by the numerator, `OrigNum`, and denominator, `OrigDen`,
with the mapping filter given by the numerator, `FTFNum`, and the
denominator, `FTFDen`. If the allpass mapping filter is not specified, then
the function returns an original filter.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);
[num, den] = iirftransf(b, a, AlpNum, AlpDen);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Here's the comparison between the filters.

iirftransf

Arguments

Variable	Description
OrigNum	Numerator of the prototype lowpass filter
OrigDen	Denominator of the prototype lowpass filter
FTFNum	Numerator of the mapping filter
FTFDen	Denominator of the mapping filter
OutNum	Numerator of the target filter
OutDen	Denominator of the target filter

See Also   zpkftransf

Purpose   Optimal IIR filter with prescribed group-delay

Syntax   ```
[num,den] = iirgrpdelay(n,f,edges,a)
[num,den] = iirgrpdelay(n,f,edges,a,w)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden)
[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden,
    tau)
[num,den,tau] = iirgrpdelay(n,f,edges,a,w)
```
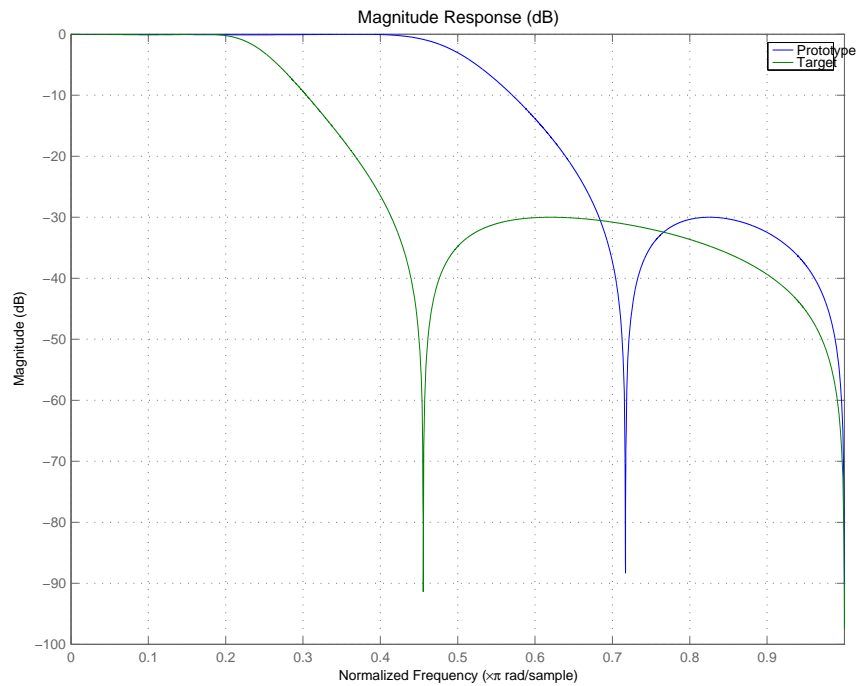
Description `[num,den] = iirgrpdelay(n,f,edges,a)` returns an allpass IIR filter of order n (n must be even) which is the best approximation to the relative group-delay response described by f and a in the least-pth sense. f is a vector of frequencies between 0 and 1 and a is specified in samples. The vector edges specifies the band-edge frequencies for multi-band designs. iirgrpdelay uses a constrained Newton-type algorithm. Always check your resulting filter using grpdelay or freqz.

`[num,den] = iirgrpdelay(n,f,edges,a,w)` uses the weights in w to weight the error. w has one entry per frequency point and must be the same length as f and a). Entries in w tell iirgrpdelay how much emphasis to put on minimizing the error in the vicinity of each specified frequency point relative to the other points.

f and a must have the same number of elements. f and a can contains more elements than the vector edges contains. This lets you use f and a to specify a filter that has any group-delay contour within each band.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius)` returns a filter having a maximum pole radius equal to radius, where 0<radius<1. radius defaults to 0.999999. Filters whose pole radius you constrain to be less than 1.0 can better retain transfer function accuracy after quantization.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p)`, where p is a two-element vector [pmin pmax], lets you determine the minimum and maximum values of p used in the least-pth algorithm. p defaults to [2

128] which yields filters very similar to the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even. If p is the string `'inspect'`, no optimization occurs. You might use this feature to inspect the initial pole/zero placement.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens)` specifies the grid density `dens` used in the optimization process. The number of grid points is `(dens*(n+1))`. The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden)` allows you to specify the initial estimate of the denominator coefficients in vector `initden`. This can be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initden`.

`[num,den] = iirgrpdelay(n,f,edges,a,w,radius,p,dens,initden,tau)` allows the initial estimate of the group delay offset to be specified by the value of `tau`, in samples.

`[num,den,tau] = iirgrpdelay(n,f,edges,a,w)` returns the resulting group delay offset. In all cases, the resulting filter has a group delay that approximates [a + tau]. Allpass filters can have only positive group delay and a non-zero value of tau accounts for any additional group delay that is needed to meet the shape of the contour specified by (f,a). The default for tau is max(a).

Hint: If the zeros or poles cluster together, your filter order may be too low or the pole radius may be too small (overly constrained). Try increasing `n` or `radius`.

For group-delay equalization of an IIR filter, compute `a` by subtracting the filter's group delay from its maximum group delay. For example,

```
[be,ae] = ellip(4,1,40,0.2);
f = 0:0.001:0.2;
g = grpdelay(be,ae,f,2);   % Equalize only the passband.
a = max(g)-g;
```

```
[num,den]=iirgrpdelay(8, f, [O 0.2], a);
```

References   Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

See Also   freqz | filter | grpdelay | iirlpnorm | iirlpnormc | zplane

iirlinphase

Purpose	Quasi-linear phase IIR filter from halfband filter specification
Syntax	`hd = design(d,'iirlinphase')` `hd = design(...,'`**`filterstructure`**`',structure)`
Description	`hd = design(d,'iirlinphase')` designs a quasi-linear phase filter `hd` specified by the filter specification object `d`.
	`hd = design(...,'`**`filterstructure`**`',structure)` returns a filter with the structure specified by `structure`. By default, the filter structure is df2sos (direct-form II with second-order sections). You can substitute one of the following strings for `structure` to specify the structure of `hd`.

Structure String	Filter Structure
df1sos	Direct-form I IIR filter with second-order sections
df2sos	Direct-form II IIR filter with second-order sections
df1tsos	Transposed direct-form I IIR filter with second-order sections
df2tsos	Transposed direct-form II IIR filter with second-order sections

Examples

Design a quasi-linear phase, minimum-order halfband IIR filter with transition width of 0.36 and stopband attenuation of at least 80 dB.

```
tw = 0.36;
ast = 80;
d = fdesign.halfband('tw,ast',tw,ast); % Transition width,
                                       % stopband attenuation.
hd = design(d,'iirlinphase');
fvtool(hd)
```

Notice the characteristic halfband nature of the ripple in the stopband. If you measure the resulting filter, you see it meets the specifications.

measure(hd)
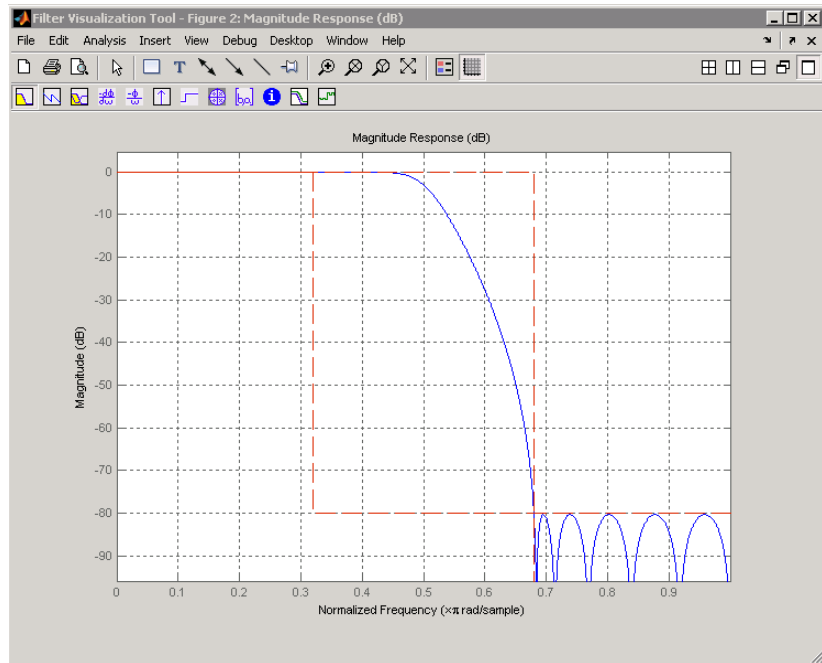
See Also fdesign.halfband

iirlp2bp

Purpose   Transform IIR lowpass filter to IIR bandpass filter

Syntax   ```
[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(B,A,Wo,Wt)
[G,AllpassNum,AllpassDen] = iirlp2bp(Hd,Wo,Wt)
```

where `Hd` is a `dfilt` object

**Description**  `[Num,Den,AllpassNum,AllpassDen] = iirlp2bp(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandpass frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency -$W_o$, at the required target frequency location, $W_{t1}$, and the second feature, originally at +$W_o$, at the new location, $W_{t2}$. It is assumed that $W_{t2}$ is greater than $W_{t1}$. This transformation implements the "DC Mobility," meaning that the Nyquist feature stays at Nyquist, but the DC feature moves to a location dependent on the selection of $W_t$s.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, $F_1$ and $F_2$, with $F_1$ preceding $F_2$. Feature $F_1$ will still precede $F_2$ after the transformation. However, the distance between $F_1$ and $F_2$ will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature: the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Real lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies.

[G,AllpassNum,AllpassDen] = iirlp2bp(Hd,Wo,Wt) returns transformed dfilt object G with a real bandpass magnitude response. The coefficients AllpassNum and AllpassDen represent the allpass mapping filter for mapping the prototype filter frequency Wo and target frequencies vector Wt. Note that in this syntax Hd is a dfilt object with a lowpass magnitude response.

**Examples**     Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b,a] = ellip(3, 0.1, 30, 0.409);
```

Create the real bandpass filter by placing the cutoff frequencies of the prototype filter at the band edge frequencies W_{t1}=0.25 and W_{t2}=0.75:

```
[num,den] = iirlp2bp(b,a,0.5,[0.25,0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b,a,num,den);
```

You can compare the results in this figure to verify the transformation.

Magnitude Response (dB)

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters,' *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also   iirftransf | allpasslp2bp | zpklp2bp

iirlp2bpc

Purpose   IIR lowpass to complex bandpass transformation

Syntax   `[Num,Den,AllpassNum,AllpassDen] = iirlp2bpc(B,A,Wo,Wt)`
                   `[G,AllpassNum,AllpassDen] = iirlp2bpc(Hd,Wo,Wt)`

                   where `Hd` is a `dfilt` object

Description   `[Num,Den,AllpassNum,AllpassDen] = iirlp2bpc(B,A,Wo,Wt)`
                   returns the numerator and denominator vectors, `Num` and `Den`
                   respectively, of the target filter transformed from the real lowpass
                   prototype by applying a first-order real lowpass to complex bandpass
                   frequency transformation.

                   It also returns the numerator, `AllpassNum`, and the denominator,
                   `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter
                   is given with a numerator specified by `B` and a denominator specified
                   by `A`.

                   This transformation effectively places one feature of an original filter,
                   located at frequency $-W_o$, at the required target frequency location, W_{t1},
                   and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is
                   assumed that W_{t2} is greater than W_{t1}.

                   Relative positions of other features of an original filter do not change in
                   the target filter. This means that it is possible to select two features of
                   an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
                   precede F_2 after the transformation. However, the distance between F_1
                   and F_2 will not be the same before and after the transformation.

                   Choice of the feature subject to the lowpass to bandpass transformation
                   is not restricted only to the cutoff frequency of an original lowpass filter.
                   In general it is possible to select any feature; e.g., the stopband edge,
                   the DC, the deep minimum in the stopband, or other ones.

                   Lowpass to bandpass transformation can also be used for transforming
                   other types of filters; for example real notch filters or resonators can be
                   doubled and positioned at two distinct desired frequencies at any place
                   around the unit circle forming a pair of complex notches/resonators.
                   This transformation can be used for designing bandpass filters for radio
                   receivers from the high-quality prototype lowpass filter.

[G,AllpassNum,AllpassDen] = iirlp2bpc(Hd,Wo,Wt) returns transformed dfilt object G with a bandpass magnitude response. The coefficients AllpassNum and AllpassDen represent the allpass mapping filter for mapping the prototype filter frequency Wo and the target frequencies vector Wt. Note that in this syntax Hd is a dfilt object with a lowpass magnitude response.

Examples   Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Move the cutoffs of the prototype filter to the new locations W_{t1}=0.25 and W_{t2}=0.75 creating a complex bandpass filter:

```
[num, den] = iirlp2bpc(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Plotting the prototype and target filters together in FVTool lets you compare the filters.

Magnitude Response (dB)

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Num	Numerator of the target filter

Variable	Description
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also   iirftransf | allpasslp2bpc | zpklp2bpc

iirlp2bs

Purpose   Transform IIR lowpass filter to IIR bandstop filter

Syntax   `[Num,Den,AllpassNum,AllpassDen] = iirlp2bs(B,A,Wo,Wt)`
                 `[G,AllpassNum,AllpassDen] = iirlp2bs(Hd,Wo,Wt)`

                 where `Hd` is a `dfilt` object

Description `[Num,Den,AllpassNum,AllpassDen] = iirlp2bs(B,A,Wo,Wt)` returns
                 the numerator and denominator vectors, `Num` and `Den` respectively, of the
                 target filter transformed from the real lowpass prototype by applying a
                 second-order real lowpass to real bandstop frequency mapping.

                 It also returns the numerator, `AllpassNum`, and the denominator,
                 `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter
                 is given with a numerator specified by `B` and a denominator specified
                 by `A`.

                 This transformation effectively places one feature of an original filter,
                 located at frequency $-W_o$, at the required target frequency location, W_{t1},
                 and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is
                 assumed that W_{t2} is greater than W_{t1}. This transformation implements
                 the "Nyquist Mobility," which means that the DC feature stays at DC,
                 but the Nyquist feature moves to a location dependent on the selection
                 of W_o and W_ts.

                 Relative positions of other features of an original filter change in the
                 target filter. This means that it is possible to select two features
                 of an original filter, F_1 and F_2, with F_1 preceding F_2. After the
                 transformation feature F_2 will precede F_1 in the target filter. However,
                 the distance between F_1 and F_2 will not be the same before and after
                 the transformation.

                 Choice of the feature subject to the lowpass to bandstop transformation
                 is not restricted only to the cutoff frequency of an original lowpass filter.
                 In general it is possible to select any feature; e.g., the stopband edge,
                 the DC, the deep minimum in the stopband, or other ones.

                 `[G,AllpassNum,AllpassDen] = iirlp2bs(Hd,Wo,Wt)` returns
                 transformed `dfilt` object `G` with a bandstop magnitude response. The
                 coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping

filter for mapping the prototype filter frequency Wo and the target frequencies vector Wt. Note that in this syntax Hd is a dfilt object with a lowpass magnitude response.

Examples   Design a prototype real IIR halfband filter using a standard elliptic approach:

```
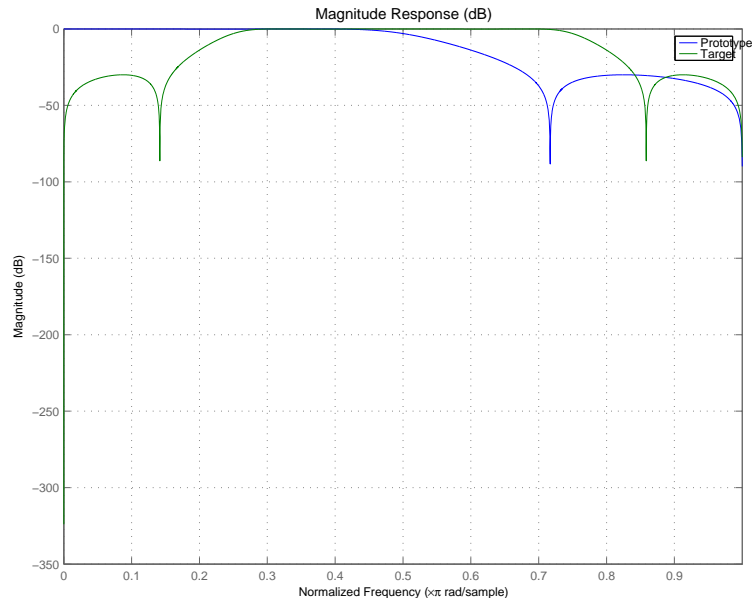[b, a] = ellip(3, 0.1, 30, 0.409);
```

Create the real bandstop filter by placing the cutoff frequencies of the prototype filter at the band edge frequencies W_{t1}=0.25 and W_{t2}=0.75:

```
[num, den] = iirlp2bs(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

iirlp2bs

With both filters plotted in the figure, you see clearly the results of the transformation.

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References
Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also   iirftransf | allpasslp2bs | zpklp2bs

iirlp2bsc

Purpose　　Transform IIR lowpass filter to IIR complex bandstop filter

Syntax　　`[Num,Den,AllpassNum,AllpassDen] = iirlp2bsc(B,A,Wo,Wt)`
`[G,AllpassNum,AllpassDen] = iirlp2bsc(Hd,Wo,Wt)`

where `Hd` is a `dfilt` object

Description　　`[Num,Den,AllpassNum,AllpassDen] = iirlp2bsc(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order real lowpass to complex bandstop frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and the denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. Additionally the transformation swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming other types of filters; e.g., real notch filters or resonators can be doubled and positioned at two distinct desired frequencies at any place around the unit circle forming a pair of complex notches/resonators. This transformation can be used for designing bandstop filters for band

attenuation or frequency equalizers, from the high-quality prototype lowpass filter.

[G,AllpassNum,AllpassDen] = iirlp2bsc(Hd,Wo,Wt) returns transformed dfilt object G with a bandstop magnitude response. The coefficients AllpassNum and AllpassDen represent the allpass mapping filter for mapping the prototype filter frequency Wo and the target frequencies vector Wt. Note that in this syntax Hd is a dfilt object with a lowpass magnitude response.

Examples Design a prototype real IIR halfband filter using a standard elliptic approach:

```
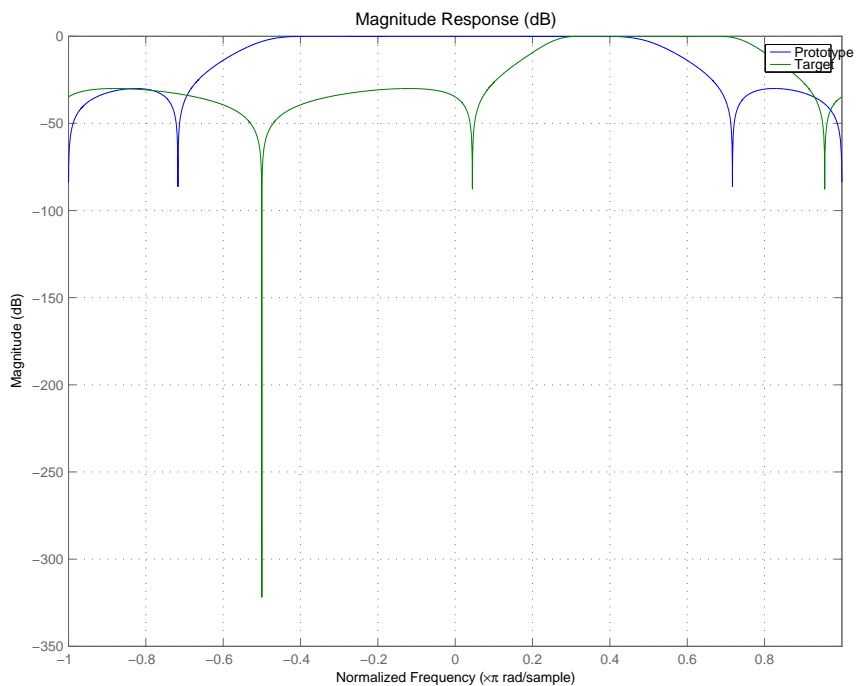[b, a] = ellip(3, 0.1, 30, 0.409);
```

Move the cutoffs of the prototype filter to the new locations W_{t1}=0.25 and W_{t2}=0.75 creating a complex bandstop filter:

```
[num, den] = iirlp2bsc(b, a, 0.5, [0.25, 0.75]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

The last command in the example plots both filters in the same window so you can compare the results.

Magnitude Response (dB)

Arguments	**Variable**	**Description**
	B	Numerator of the prototype lowpass filter
	A	Denominator of the prototype lowpass filter
	Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
	Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

Variable	Description
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also   iirftransf | allpasslp2bsc | zpklp2bsc

iirlp2hp

Purpose	Transform lowpass IIR filter to highpass filter

Syntax

```
[num,den] = iirlp2hp(b,a,wc,wd)
[G,AllpassNum,AllpassDen] = iirlp2hp(Hd,Wo,Wt)
```

where Hd is a dfilt object

Description

[num,den] = iirlp2hp(b,a,wc,wd) with input arguments b and a, the numerator and denominator coefficients (zeros and poles) for a lowpass IIR filter, iirlp2bp transforms the magnitude response from lowpass to highpass. num and den return the coefficients for the transformed highpass filter. For wc, enter a selected frequency from your lowpass filter. You use the chosen frequency to define the magnitude response value you want in the highpass filter. Enter one frequency for the highpass filter — the value that defines the location of the transformed point — in wd. Note that all frequencies are normalized between zero and one. Notice also that the filter order does not change when you transform to a highpass filter.

When you select wc and designate wd, the transformation algorithm sets the magnitude response at the wd values of your bandstop filter to be the same as the magnitude response of your lowpass filter at wc. Filter performance between the values in wd is not specified, except that the stopband retains the ripple nature of your original lowpass filter and the magnitude response in the stopband is equal to the peak response of your lowpass filter. To accurately specify the filter magnitude response across the stopband of your bandpass filter, use a frequency value from within the stopband of your lowpass filter as wc. Then your bandstop filter response is the same magnitude and ripple as your lowpass filter stopband magnitude and ripple.

The fact that the transformation retains the shape of the original filter is what makes this function useful. If you have a lowpass filter whose characteristics, such as rolloff or passband ripple, particularly meet your needs, the transformation function lets you create a new filter with the same characteristic performance features, but in a highpass version. Without designing the highpass filter from the beginning.

In some cases transforming your filter may cause numerical problems, resulting in incorrect conversion to the highpass filter. Use fvtool to verify the response of your converted filter.

[G,AllpassNum,AllpassDen] = iirlp2hp(Hd,Wo,Wt) returns transformed dfilt object G with a highpass magnitude response. The coefficients AllpassNum and AllpassDen represent the allpass mapping filter for mapping the prototype filter frequency Wo and the target frequencies vector Wt. Note that in this syntax Hd is a dfilt object with a lowpass magnitude response.

Examples   This example transforms an IIR filter from lowpass to high pass by moving the magnitude response at one frequency in the source filter to a new location in the transformed filter. To generate a highpass filter whose passband flattens out at 0.4, select the frequency in the lowpass filter where the passband starts to rolloff (wc = 0.0175) and move it to the new location at wd = 0.4.

```
[b,a] = iirlpnorm(10,6,[0 0.0175 0.02 0.0215 0.025 1],...
[0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0],...
[1 1 1 1 10 10]);
wc = 0.0175;
wd = 0.4;
[num,den] = iirlp2hp(b,a,wc,wd);
fvtool(b,a,num,den);
```

iirlp2hp

In the figure showing the magnitude responses for the two filters, the transition band for the highpass filter is essentially the mirror image of the transition for the lowpass filter from 0.0175 to 0.025, stretched out over a wider frequency range. In the passbands, the filter share common ripple characteristics and magnitude.

References   Mitra, Sanjit K., *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

See Also   iirlp2bp | iirlp2bs | iirlp2lp | firlp2lp | firlp2hp

Purpose   Transform lowpass IIR filter to different lowpass filter

Syntax   ```
[num,den] = iirlp2hp(b,a,wc,wd)
[G,AllpassNum,AllpassDen] = iirlp2lp(Hd,Wo,Wt)
```
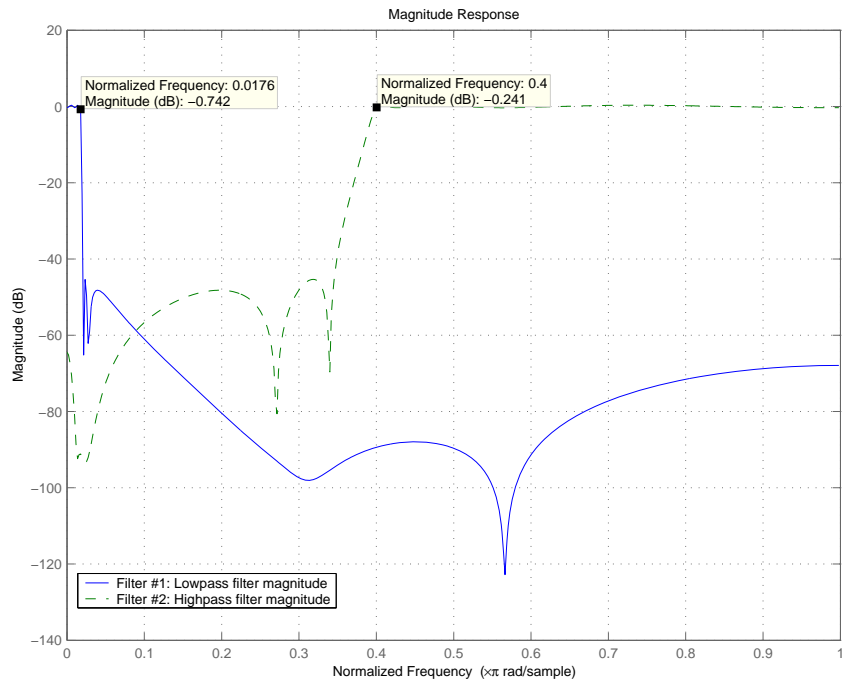
where `Hd` is a `dfilt` object

Description `[num,den] = iirlp2hp(b,a,wc,wd)` with input arguments `b` and `a`, the numerator and denominator coefficients (zeros and poles) for a lowpass IIR filter, `iirlp2bp` transforms the magnitude response from lowpass to highpass. `num` and `den` return the coefficients for the transformed highpass filter. For `wc`, enter a selected frequency from your lowpass filter. You use the chosen frequency to define the magnitude response value you want in the highpass filter. Enter one frequency for the highpass filter — the value that defines the location of the transformed point — in `wd`. Note that all frequencies are normalized between zero and one. Notice also that the filter order does not change when you transform to a highpass filter.

When you select `wc` and designate `wd`, the transformation algorithm sets the magnitude response at the `wd` values of your bandstop filter to be the same as the magnitude response of your lowpass filter at `wc`. Filter performance between the values in `wd` is not specified, except that the stopband retains the ripple nature of your original lowpass filter and the magnitude response in the stopband is equal to the peak response of your lowpass filter. To accurately specify the filter magnitude response across the stopband of your bandpass filter, use a frequency value from within the stopband of your lowpass filter as `wc`. Then your bandstop filter response is the same magnitude and ripple as your lowpass filter stopband magnitude and ripple.

The fact that the transformation retains the shape of the original filter is what makes this function useful. If you have a lowpass filter whose characteristics, such as rolloff or passband ripple, particularly meet your needs, the transformation function lets you create a new filter with the same characteristic performance features, but in a highpass version. Without designing the highpass filter from the beginning.

In some cases transforming your filter may cause numerical problems, resulting in incorrect conversion to the highpass filter. Use `fvtool` to verify the response of your converted filter.

`[G,AllpassNum,AllpassDen] = iirlp2lp(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with a lowpass magnitude response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples   This example transforms an IIR filter from lowpass to high pass by moving the magnitude response at one frequency in the source filter to a new location in the transformed filter. To generate a lowpass filter whose passband extends out to 0.2, select the frequency in the lowpass filter where the passband starts to rolloff (`wc = 0.0175`) and move it to the new location at `wd = 0.2`.

```
[b,a] = iirlpnorm(10,6,[0 0.0175 0.02 0.0215 0.025 1],...
[0 0.0175 0.02 0.0215 0.025 1],[1 1 0 0 0 0],...
[1 1 1 1 10 10]);
wc = 0.0175;
wd = 0.2;
[num,den] = iirlp2lp(b,a,wc,wd);
fvtool(b,a,num,den);
```

Moving the edge of the passband from 0.0175 to 0.2 results in a new lowpass filter whose peak response in-band is the same as the original filter: same ripple, same absolute magnitude.
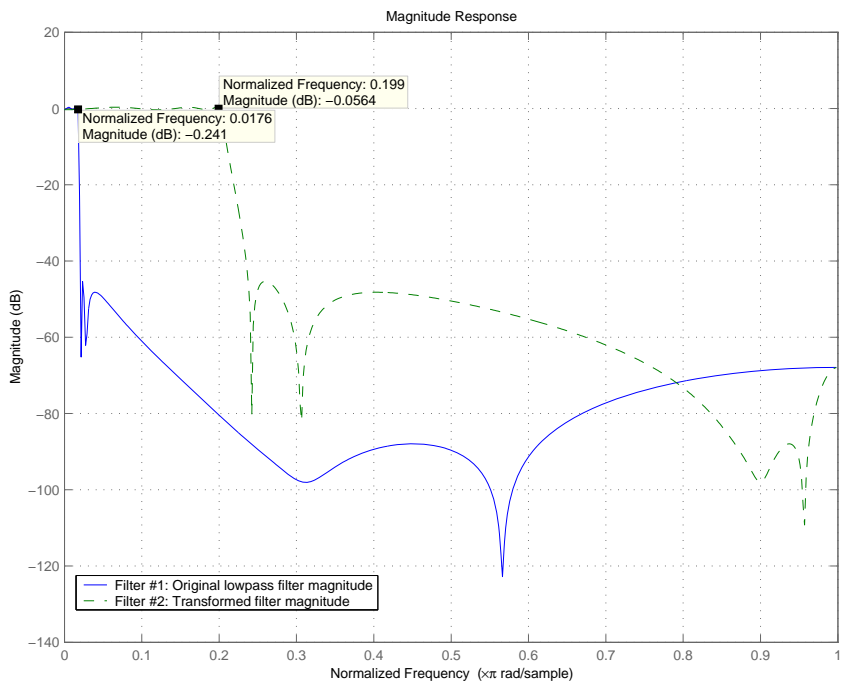
The rolloff is slightly less steep and the stopband profiles are the same for both filters; the new filter stopband is a "stretched" version of the original, as is the passband of the new filter.

References   Mitra, Sanjit K, *Digital Signal Processing. A Computer-Based Approach*, Second Edition, McGraw-Hill, 2001.

See Also   iirlp2bp | iirlp2bs | iirlp2hp | firlp2lp | firlp2hp

iirlp2mb

Purpose

Transform IIR lowpass filter to IIR M-band filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen]=iirlp2mb(B,A,Wo,Wt,Pass)
[G,AllpassNum,AllpassDen] = iirlp2mb(Hd,Wo,Wt)
[G,AllpassNum,AllpassDen] = iirlp2mb(...,Pass)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirlp2mb(B,A,Wo,Wt)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying an Mth-order real lowpass to real multiple bandpass frequency mapping. By default the DC feature is kept at its original location.

`[Num,Den,AllpassNum,AllpassDen]=iirlp2mb(B,A,Wo,Wt,Pass)` allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is movable.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

This transformation effectively places one feature of an original filter, located at frequency W_o, at the required target frequency locations, $W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter do not change in the target filter. It is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

`[G,AllpassNum,AllpassDen] = iirlp2mb(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with an IIR real M-band filter frequency response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

`[G,AllpassNum,AllpassDen] = iirlp2mb(...,Pass)` returns transformed `dfilt` object `G` with an IIR real M-band filter frequency response. This syntax allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

Examples   Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Example 1

Create the real multiband filter with two passbands:

```
[num1, den1] = iirlp2mb(b, a, 0.5, [2 4 6 8]/10);
[num2, den2] = iirlp2mb(b, a, 0.5, [2 4 6 8]/10, 'pass');
```

The second code snippet uses the `pass` option to select the Nyquist mobility option. In this case the resulting filter is the same.

Example 2

Create the real multiband filter with two stopbands:

```
[num3, den3] = iirlp2mb(b, a, 0.5, [2 4 6 8]/10, 'stop');
```

Verify the result by comparing the prototype filter with target filters:

```
fvtool(b, a, num1, den1, num2, den2, num3, den3);
```

Combining all of the filters, prototypes and targets, on one figure makes comparing them straightforward. Passbands for the filters in example 1 appear separately in the figure, although they overlap to a degree that makes them hard to identify — they have identical coefficients.

| Magnitude Response (dB) |

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default
Num	Numerator of the target filter

iirlp2mb

Variable	Description
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References
Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems," *MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

Feyh, G., J.C. Franchitti and C.T. Mullis, "All-pass filter interpolation and frequency transformation problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

Mullis, C.T. and R. A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Mass., Addison-Wesley, 1987.

Feyh, G., W.B. Jones and C.T. Mullis, "An extension of the Schur Algorithm for frequency transformations," *Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

See Also   iirftransf | allpasslp2mb | zpklp2mb

Purpose   Transform IIR lowpass filter to IIR complex M-band filter

Syntax   [Num,Den,AllpassNum,AllpassDen] = iirlp2mbc(B,A,Wo,Wc)
                   [G,AllpassNum,AllpassDen] = iirlp2mbc(Hd,Wo,Wt)

                   where Hd is a `dfilt` object

Description   [Num,Den,AllpassNum,AllpassDen] = iirlp2mbc(B,A,Wo,Wc)
                   returns the numerator and denominator vectors, Num and Den
                   respectively, of the target filter transformed from the real lowpass
                   prototype by applying an Mth-order real lowpass to complex
                   multibandpass frequency transformation.

                   It also returns the numerator, AllpassNum, and the denominator,
                   AllpassDen, of the allpass mapping filter. The prototype lowpass filter
                   is given with a numerator specified by B and a denominator specified
                   by A.

                   This transformation effectively places one feature of an original filter,
                   located at frequency W_o, at the required target frequency locations,
                   $W_{t1},...,W_{tM}$.

                   Relative positions of other features of an original filter do not change in
                   the target filter. This means that it is possible to select two features of
                   an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
                   precede F_2 after the transformation. However, the distance between F_1
                   and F_2 will not be the same before and after the transformation.

                   Choice of the feature subject to this transformation is not restricted
                   to the cutoff frequency of an original lowpass filter. In general it is
                   possible to select any feature; e.g., the stopband edge, the DC, the deep
                   minimum in the stopband, or other ones.

                   This transformation can also be used for transforming other types of
                   filters; e.g., notch filters or resonators can be easily replicated at a
                   number of required frequency locations. A good application would be
                   an adaptive tone cancellation circuit reacting to the changing number
                   and location of tones.

[G,AllpassNum,AllpassDen] = iirlp2mbc(Hd,Wo,Wt) returns transformed `dfilt` object G with an IIR complex M-band filter frequency response. The coefficients AllpassNum and AllpassDen represent the allpass mapping filter for mapping the prototype filter frequency Wo and the target frequencies vector Wt. Note that in this syntax Hd is a `dfilt` object with a lowpass magnitude response.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Now create a complex multiband filter with two passbands:

```
[num1, den1] = iirlp2mbc(b, a, 0.5, [2 4 6 8]/10);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num1, den1);
```

Magnitude Response (dB)
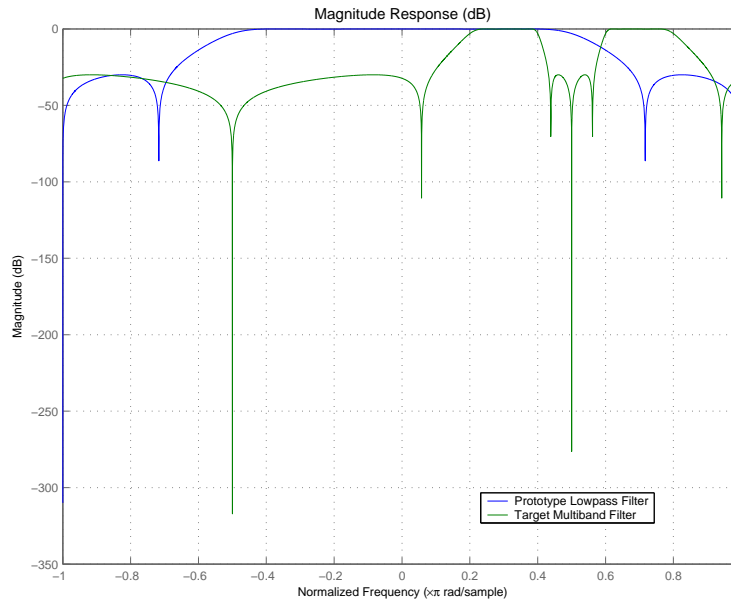
You see in the figure that `iirlp2mbc` replicates the desired feature at 0.5 in the lowpass filter at four locations in the multiband filter.

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter.
A	Denominator of the prototype lowpass filter.
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wc	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Num	Numerator of the target filter.

iirlp2mbc

Variable	Description
Den	Denominator of the target filter.
AllpassNum	Numerator of the mapping filter.
AllpassDen	Denominator of the mapping filter.

See Also   iirftransf | allpasslp2mbc | zpklp2mbc

Purpose   Transform IIR lowpass filter to IIR complex N-point filter

Syntax   [Num,Den,AllpassNum,AllpassDen] = iirlp2xc(B,A,Wo,Wt)
                   [G,AllpassNum,AllpassDen] = iirlp2xc(Hd,Wo,Wt)

                   where Hd is a dfilt object

Description   [Num,Den,AllpassNum,AllpassDen] = iirlp2xc(B,A,Wo,Wt) returns
                   the numerator and denominator vectors, Num and Den respectively, of the
                   target filter transformed from the real lowpass prototype by applying an
                   Nth-order real lowpass to complex multipoint frequency transformation.

                   It also returns the numerator, AllpassNum, and the denominator,
                   AllpassDen, of the allpass mapping filter. The prototype lowpass filter
                   is given with a numerator specified by B and a denominator specified
                   by A.

                   Parameter N also specifies the number of replicas of the prototype
                   filter created around the unit circle after the transformation. This
                   transformation effectively places N features of an original filter, located
                   at frequencies W_{o1},...,W_{oN}, at the required target frequency locations,
                   W_{t1},...,W_{tM}.

                   Relative positions of other features of an original filter are the same
                   in the target filter for the Nyquist mobility and are reversed for the
                   DC mobility. For the Nyquist mobility this means that it is possible to
                   select two features of an original filter, F_1 and F_2, with F_1 preceding F_2.
                   Feature F_1 will still precede F_2 after the transformation. However,
                   the distance between F_1 and F_2 will not be the same before and after
                   the transformation. For DC mobility feature F_2 will precede F_1 after
                   the transformation.

                   Choice of the feature subject to this transformation is not restricted to
                   the cutoff frequency of an original lowpass filter. In general it is possible
                   to select any feature; e.g., a stopband edge, DC, the deep minimum in
                   the stopband, or other ones. The only condition is that the features
                   must be selected in such a way that when creating N bands around the
                   unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

`[G,AllpassNum,AllpassDen] = iirlp2xc(Hd,Wo,Wt)` returns transformed `dfilt` object G with an IIR complex N-point filter frequency response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency Wo and the target frequencies vector Wt. Note that in this syntax Hd is a `dfilt` object with a lowpass magnitude response.

Examples   Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Create the complex bandpass filter from the real lowpass filter:

```
[num, den] = iirlp2xc(b, a, [-0.5 0.5], [-0.25 0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Reviewing the coefficients and the figure produced by the example shows that the target filter has complex coefficients and is indeed a bandpass filter as expected.

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter.
A	Denominator of the prototype lowpass filter.
Wo	Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Num	Numerator of the target filter.

iirlp2xc

Variable	Description
Den	Denominator of the target filter.
AllpassNum	Numerator of the mapping filter.
AllpassDen	Denominator of the mapping filter.

See Also   iirftransf | allpasslp2xc | zpklp2xc

Purpose　　　Transform IIR lowpass filter to IIR real N-point filter

Syntax　　　　```
[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt,Pass)
[G,AllpassNum,AllpassDen] = iirlp2bpc(Hd,Wo,Wt),
where Hd is a dfilt object
[G,AllpassNum,AllpassDen] = iirlp2bpc(...,Pass)
```

**Description**　　[Num,Den,AllpassNum,AllpassDen] = iirlp2xn(B,A,Wo,Wt) returns
the numerator and denominator vectors, Num and Den respectively,
of the target filter transformed from the real lowpass prototype by
applying an Nth-order real lowpass to real multipoint frequency
transformation, where N is the number of features being mapped. By
default the DC feature is kept at its original location.

[Num,Den,AllpassNum,AllpassDen]= iirlp2xn(B,A,Wo,Wt,Pass)
allows you to specify an additional parameter, Pass, which chooses
between using the "DC Mobility" and the "Nyquist Mobility." In the
first case the Nyquist feature stays at its original location and the DC
feature is free to move. In the second case the DC feature is kept at an
original frequency and the Nyquist feature is allowed to move.

It also returns the numerator, AllpassNum, and the denominator,
AllpassDen, of the allpass mapping filter. The prototype lowpass
filter is given with the numerator specified by B and the denominator
specified by A.

Parameter N also specifies the number of replicas of the prototype
filter created around the unit circle after the transformation. This
transformation effectively places N features of an original filter, located
at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations,
$W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same
in the target filter for the Nyquist mobility and are reversed for the
DC mobility. For the Nyquist mobility this means that it is possible to
select two features of an original filter, $F_1$ and $F_2$, with $F_1$ preceding $F_2$.
Feature $F_1$ will still precede $F_2$ after the transformation. However,
the distance between $F_1$ and $F_2$ will not be the same before and after

the transformation. For DC mobility feature $F_2$ will precede $F_1$ after the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating `N` bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

`[G,AllpassNum,AllpassDen] = iirlp2xn(Hd,Wo,Wt)` returns transformed `dfilt` object `G` with an IIR real N-point filter frequency response. The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

`[G,AllpassNum,AllpassDen] = iirlp2xn(...,Pass)` returns transformed `dfilt` object `G` with an IIR real N-point filter frequency response. This syntax allows you to specify an additional parameter, `Pass`, which chooses between using the "DC Mobility" and the "Nyquist Mobility." In the first case the Nyquist feature stays at its original location and the DC feature is free to move. In the second case the DC feature is kept at an original frequency and the Nyquist feature is allowed to move.

The coefficients `AllpassNum` and `AllpassDen` represent the allpass mapping filter for mapping the prototype filter frequency `Wo` and the target frequencies vector `Wt`. Note that in this syntax `Hd` is a `dfilt` object with a lowpass magnitude response.

**Examples**    Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Move the cutoffs of the prototype filter to the new locations W_{t1}=0.25 and W_{t2}=0.75 creating a real bandpass filter:

```
[num, den] = iirlp2xn(b, a, [-0.5 0.5], [0.25 0.75], ...
pass');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

iirlp2xn has created the desired bandpass filter with the cutoff locations specified in the command.

Magnitude Response in dB

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency values to be transformed from the prototype filter
Wt	Desired frequency locations in the transformed target filter
Pass	Choice ('pass'/'stop') of passband/stopband at DC, 'pass' being the default
Num	Numerator of the target filter

Variable	Description
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Cain, G.D., A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

Krukowski, A., G.D. Cain and I. Kale, "Custom designed high-order frequency transformations for IIR filters," *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

See Also   iirftransf | allpasslp2xn | zpklp2xn

iirlpnorm

Purpose　　　　Least P-norm optimal IIR filter

Syntax　　　　　```
[num,den] = iirlpnorm(n,d,f,edges,a)
[num,den] = iirlpnorm(n,d,f,edges,a,w)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens)
[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens,initnum,initden)
```

Description　　`[num,den] = iirlpnorm(n,d,f,edges,a)` returns a filter having
a numerator order `n` and denominator order `d` which is the best
approximation to the desired frequency response described by `f` and
`a` in the least-pth sense. The vector `edges` specifies the band-edge
frequencies for multi-band designs. An unconstrained quasi-Newton
algorithm is employed and any poles or zeros that lie outside of the
unit circle are reflected back inside. `n` and `d` should be chosen so that
the zeros and poles are used effectively. See the "Hints" on page 7-917
section. Always use `freqz` to check the resulting filter.

`[num,den] = iirlpnorm(n,d,f,edges,a,w)` uses the weights in `w`
to weight the error. `w` has one entry per frequency point (the same
length as `f` and `a`) which tells `iirlpnorm` how much emphasis to put
on minimizing the error in the vicinity of each frequency point relative
to the other points. `f` and `a` must have the same number of elements,
which may exceed the number of elements in `edges`. This allows for
the specification of filters having any gain contour within each band.
The frequencies specified in `edges` must also appear in the vector `f`.
For example,

```
[num,den] = iirlpnorm(5,12,[0 .15 .4 .5 1],[0 .4 .5 1],...
[1 1.6 1 0 0],[1 1 1 10 10])
```

is a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p)` where `p` is a
two-element vector `[pmin pmax]` allows for the specification of the
minimum and maximum values of `p` used in the least-pth algorithm.
Default is [2 128] which essentially yields the L-infinity, or Chebyshev,
norm. Pmin and pmax should be even. If `p` is the string `'inspect'`, no

optimization will occur. This can be used to inspect the initial pole/zero placement.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is `(dens*(n+d+1))`. The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirlpnorm(n,d,f,edges,a,w,p,dens,initnum,initden)` allows for the specification of the initial estimate of the filter numerator and denominator coefficients in vectors `initnum` and `initden`. This may be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initnum` and `initden`.

Hints
- This is a weighted least-pth optimization.

- Check the radii and locations of the poles and zeros for your filter. If the zeros are on the unit circle and the poles are well inside the unit circle, try increasing the order of the numerator or reducing the error weighting in the stopband.

- Similarly, if several poles have a large radii and the zeros are well inside of the unit circle, try increasing the order of the denominator or reducing the error weighting in the passband.

References   Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

See Also   iirlpnormc | filter | freqz | iirgrpdelay | zplane

iirlpnormc

Purpose　　　Constrained least Pth-norm optimal IIR filter

Syntax　　　　`[num,den] = iirlpnormc(n,d,f,edges,a)`
`[num,den] = iirlpnormc(n,d,f,edges,a,w)`
`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)`
`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)`
`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens)`
`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens,...`
`initnum,initden)`
`[num,den,err] = iirlpnormc(...)`
`[num,den,err,sos,g] = iirlpnormc(...)`

Description　　`[num,den] = iirlpnormc(n,d,f,edges,a)` returns a filter having
numerator order `n` and denominator order `d` which is the best
approximation to the desired frequency response described by `f` and
`a` in the least-pth sense. The vector `edges` specifies the band-edge
frequencies for multi-band designs. A constrained Newton-type
algorithm is employed. `n` and `d` should be chosen so that the zeros and
poles are used effectively. See the Hints section. Always check the
resulting filter using `fvtool`.

`[num,den] = iirlpnormc(n,d,f,edges,a,w)` uses the weights in `w`
to weight the error. `w` has one entry per frequency point (the same
length as `f` and `a`) which tells `iirlpnormc` how much emphasis to put
on minimizing the error in the vicinity of each frequency point relative
to the other points. `f` and `a` must have the same number of elements,
which can exceed the number of elements in `edges`. This allows for
the specification of filters having any gain contour within each band.
The frequencies specified in `edges` must also appear in the vector `f`.
For example,

`[num,den] = iirlpnormc(5,5,[0 .15 .4 .5 1],[0 .4 .5 1],...`
`[1 1.6 1 0 0],[1 1 1 10 10])`

designs a lowpass filter with a peak of 1.6 within the passband.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius)` returns a filter
having a maximum pole radius of `radius` where 0<`radius`<1. `radius`

defaults to 0.999999. Filters that have a reduced pole radius may retain better transfer function accuracy after you quantize them.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p)` where `p` is a two-element vector [`pmin pmax`] allows for the specification of the minimum and maximum values of `p` used in the least-pth algorithm. Default is [2 128] which essentially yields the L-infinity, or Chebyshev, norm. `pmin` and `pmax` should be even. If `p` is the string `'inspect'`, no optimization will occur. This can be used to inspect the initial pole/zero placement.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens)` specifies the grid density `dens` used in the optimization. The number of grid points is (`dens*(n+d+1)`). The default is 20. `dens` can be specified as a single-element cell array. The grid is not equally spaced.

`[num,den] = iirlpnormc(n,d,f,edges,a,w,radius,p,dens,...initnum,initden)` allows for the specification of the initial estimate of the filter numerator and denominator coefficients in vectors `initnum` and `initden`. This may be useful for difficult optimization problems. The pole-zero editor in Signal Processing Toolbox software can be used for generating `initnum` and `initden`.

`[num,den,err] = iirlpnormc(...)` returns the least-Pth approximation error `err`.

`[num,den,err,sos,g] = iirlpnormc(...)` returns the second-order section representation in the matrix SOS and gain G. For numerical reasons you may find SOS and G beneficial in some cases.

Hints
- This is a weighted least-pth optimization.

- Check the radii and location of the resulting poles and zeros.

- If the zeros are all on the unit circle and the poles are well inside of the unit circle, try increasing the order of the numerator or reducing the error weighting in the stopband.

- Similarly, if several poles have a large radius and the zeros are well inside of the unit circle, try increasing the order of the denominator or reducing the error weight in the passband.

- If you reduce the pole radius, you might need to increase the order of the denominator.

The message

```
Poorly conditioned matrix. See the "help" file.
```

indicates that iirlpnormc cannot accurately compute the optimization because either:

1 The approximation error is extremely small (try reducing the number of poles or zeros — refer to the hints above).

2 The filter specifications have huge variation, such as a=[1 1e9 0 0].

Examples

This example returns a lowpass filter whose pole radius is constrained to 0.8

```
[b,a,err,s,g] = iirlpnormc(6,6,[0 .4 .5 1],[0 .4 .5 1],...
[1 1 0 0],[1 1 1 1],.8);
hd = dfilt.df1sos(s,g); % Construct second-order sections filter.
fvtool(hd); % View filter's magnitude response
```

From the magnitude response shown here you see the lowpass nature of the filter. The pole/zero plot following shows that the poles are constrained to 0.8 as specified in the command.

Magnitude Response (dB)

Pole/Zero Plot
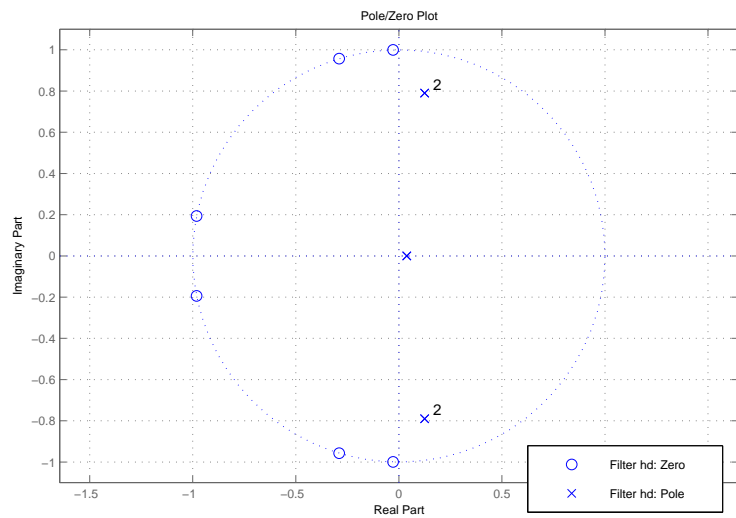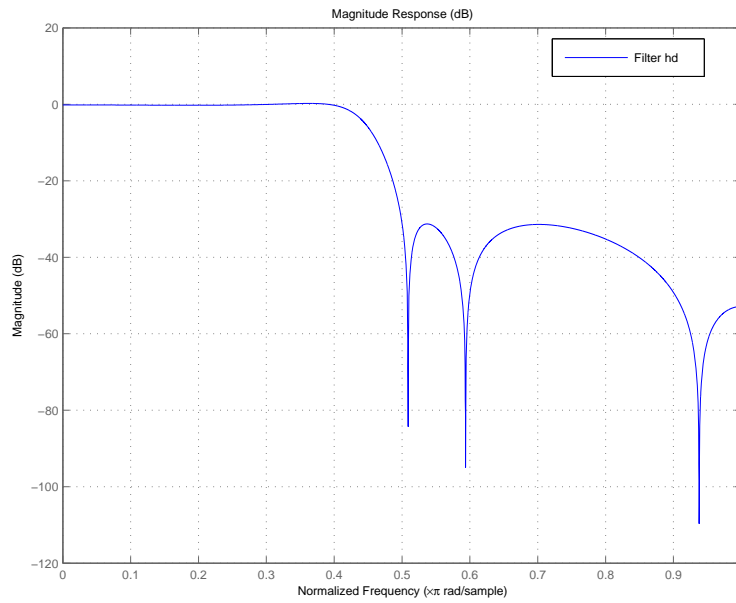
iirlpnormc

References   Antoniou, A., *Digital Filters: Analysis, Design, and Applications*, Second Edition, McGraw-Hill, Inc. 1993.

See Also   `freqz` | `filter` | `iirgrpdelay` | `iirlpnorm` | `zplane`

Purpose   RLS IIR filter from specification object

Syntax   ```
hd = design(d,'iirls')
hd = design(d,'iirls',designoption,value,designoption,value,
    ...)
```

**Description**  `hd = design(d,'iirls')` designs a least-squares filter specified by the filter specification object `d`.

---

**Note** The `iirls` algorithm might not be well behaved in all cases. Experience is your best guide to determining if the resulting filter meets your needs. When you use `iirls` to design a filter, review the filter carefully to ensure that it is appropriate for your use.

---

```
hd =
design(d,'iirls',designoption,value,designoption,value,...)
```
returns a least-squares IIR filter where you specify design options as input arguments.

To determine the available design options, use `designopts` with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using `iirls`, refer to the command line help system. For example, to get specific information about using `iirls` with `d`, the specification object, enter the following at the MATLAB prompt.

```
help(d,'iirls')
```

Examples   Starting from an arbitrary magnitude and phase design object `d`, generate a complex bandpass filter of order = 5. To make the example a little easier to do, use the default values for F, and H, the frequency vector and the complex desired frequency response.

```
d = fdesign.arbmagnphase('N,F,H',5);
d =

            Response: 'Arbitrary Magnitude and Phase'
       Specification: 'N,F,H'
         Description: {'Filter Order';'Frequency Vector';'
                       Complex Desired Frequency Response'
 NormalizedFrequency: true
          FilterOrder: 5
          Frequencies: [1x655 double]
         FreqResponse: [1x655 double]


design(d,'iirls'); % Opens FVTool to show the filter.
```

Displaying both the phase and magnitude response in FVTool shows you the filter.

See Also   fdesign.arbmag | fdesign.arbmagnphase | firls

iirnotch

Purpose	Second-order IIR notch filter
Syntax	[num,den] = iirnotch(w0,bw)
	[num,den] = iirnotch(w0,bw,ab)

Description   [num,den] = iirnotch(w0,bw) turns a digital notching filter with the notch located at w0, and with the bandwidth at the -3 dB point set to bw. To design the filter, w0 must meet the condition 0.0 w0 1.0, where 1.0 corresponds to π adians per sample in the frequency range.

The quality factor (Q factor) q for the filter is related to the filter bandwidth by q w0/bw where ω_0 is w0, the frequency to remove from the signal.

[num,den] = iirnotch(w0,bw,ab) returns a digital notching filter whose bandwidth, bw, is specified at a level of -ab decibels. Including the optional input argument ab lets you specify the magnitude response bandwidth at a level that is not the default -3 dB point, such as -6 dB or 0 dB.

Examples   Design and plot an IIR notch filter that removes a 60 Hz tone (f0) from a signal at 300 Hz (fs). For this example, set the Q factor for the filter to 35 and use it to specify the filter bandwidth:

```
wo = 60/(300/2);   bw = wo/35;
[b,a] = iirnotch(wo,bw);
fvtool(b,a);
```

Shown in the next plot, the notch filter has the desired bandwidth with the notch located at 60 Hz, or 0.4π radians per sample. Compare this plot to the comb filter plot shown on the reference page for iircomb.

See Also   firgr | iircomb | iirpeak

iirpeak

Purpose	Second-order IIR peak or resonator filter
Syntax	`[num,den] = iirpeak(w0,bw)` `[num,den] = iirpeak(w0,bw,ab)`

Description `[num,den] = iirpeak(w0,bw)` turns a second-order digital peaking filter with the peak located at `w0`, and with the bandwidth at the +3 dB point set to `bw`. To design the filter, `w0` must meet the condition $0.0 < w0 < 1.0$, where 1.0 corresponds to π radians per sample in the frequency range.

The quality factor (Q factor) q for the filter is related to the filter bandwidth by `q = w0/bw` where ω_0 is `w0` the signal frequency to boost.

`[num,den] = iirpeak(w0,bw,ab)` returns a digital peaking filter whose bandwidth, `bw`, is specified at a level of +`ab` decibels. Including the optional input argument `ab` lets you specify the magnitude response bandwidth at a level that is not the default +3 dB point, such as +6 dB or 0 dB.

Examples Design and plot an IIR peaking filter that boosts the frequency at 1.75 Khz in a signal and has bandwidth of 500 Hz at the -3 dB point:

```
fs = 10000; wo = 1750/(fs/2);  bw = 500/(fs/2);
[b,a] = iirpeak(wo,bw);
fvtool(b,a);
```

Shown in the next plot, the peak filter has the desired gain and bandwidth at 1.75 KHz.

See Also   firgr | iircomb | iirnotch

iirpowcomp

Purpose   Power complementary IIR filter

Syntax   ```
[bp,ap] = iirpowcomp(b,a)
[bp,ap,c] = iirpowcomp(b,a)
```
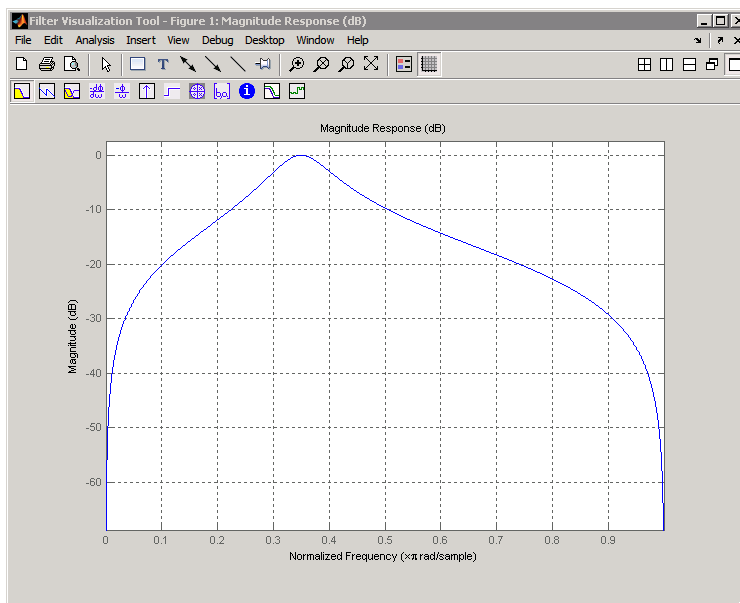
Description   `[bp,ap] = iirpowcomp(b,a)` returns the coefficients of the power complementary IIR filter g(z) = bp(z)/ap(z) in vectors `bp` and `ap`, given the coefficients of the IIR filter h(z) = b(z)/a(z) in vectors `b` and `a`. `b` must be symmetric (Hermitian) or antisymmetric (antihermitian) and of the same length as `a`. The two power complementary filters satisfy the relation

$$|H(w)|^2 + |G(w)|^2 = 1.$$

`[bp,ap,c] = iirpowcomp(b,a)` where `c` is a complex scalar of magnitude = 1, forces `bp` to satisfy the generalized hermitian property

`conj(bp(end:-1:1)) = c*bp`.

When `c` is omitted, it is chosen as follows:

- When `b` is real, chooses C as 1 or -1, whichever yields `bp` real

- When `b` is complex, C defaults to 1

`ap` is always equal to `a`.

Examples   ```
[b,a]=cheby1(10,.5,.4);
[bp,ap]=iirpowcomp(b,a);
Hd1 = dfilt.df2(b,a);
Hd2 = dfilt.df2(bp,ap);
hfvt = fvtool([Hd1,Hd2],'MagnitudeDisplay','Magnitude');
legend(hfvt,'Original Filter','Power Complementary Version');
```

The next figure presents the results of applying `iirpowcomp` to the Chebyshev filter — the power complementary version of the original filter.

**See Also**     tf2ca | tf2cl | ca2tf | cl2tf

# iirrateup

**Purpose**

Upsample IIR filter by integer factor

**Syntax**

`[Num,Den,AllpassNum,AllpassDen] = iirrateup(B,A,N)`

**Description**

`[Num,Den,AllpassNum,AllpassDen] = iirrateup(B,A,N)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter being transformed from any prototype by applying an `N`th-order rateup frequency transformation, where `N` is the upsample ratio. Transformation creates `N` equal replicas of the prototype filter frequency response.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with a numerator specified by `B` and a denominator specified by `A`.

The relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, $F_1$ and $F_2$, with $F_1$ preceding $F_2$. Feature $F_1$ will still precede $F_2$ after the transformation. However, the distance between $F_1$ and $F_2$ will not be the same before and after the transformation.

**Examples**

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
[num, den] = iirrateup(b, a, 4);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

As shown in the figure produced by FVTool, the transformed filter appears as expected.

Magnitude Response in dB

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
N	Frequency multiplication ratio
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

iirrateup

See Also   iirtransf | allpassrateup | zpkrateup

Purpose   Shift frequency response of IIR filter

Syntax   `[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)`

Description `[Num,Den,AllpassNum,AllpassDen] = iirshift(B,A,Wo,Wt)` returns
the numerator and denominator vectors, `Num` and `Den` respectively,
of the target filter transformed from the real lowpass prototype by
applying a second-order real shift frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator of the
allpass mapping filter, `AllpassDen`. The prototype lowpass filter is
given with the numerator specified by `B` and the denominator specified
by `A`.

This transformation places one selected feature of an original filter
located at frequency W_o to the required target frequency location, W_t.
This transformation implements the "DC Mobility," which means that
the Nyquist feature stays at Nyquist, but the DC feature moves to a
location dependent on the selection of W_o and W_t.

Relative positions of other features of an original filter do not change in
the target filter. This means that it is possible to select two features of
an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
precede F_2 after the transformation. However, the distance between F_1
and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the real shift transformation is not
restricted to the cutoff frequency of an original lowpass filter. In general
it is possible to select any feature; e.g., the stopband edge, the DC, the
deep minimum in the stopband, or other ones.

This transformation can also be used for transforming other types of
filters; e.g., notch filters or resonators can change their position in a
simple way without designing them from the beginning.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Perform the real frequency shift by defining where the selected feature of the prototype filter, originally at W_o=0.5, should be placed in the target filter, W_t=0.75:

```
Wo = 0.5; Wt = 0.75;
[num, den] = iirshift(b, a, Wo, Wt);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

Shifting the specified feature from the prototype to the target generates the response shown in the figure.

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

See Also   iirftransf | allpassshift | zpkshift

iirshiftc

Purpose

Shift frequency response of IIR complex filter

Syntax

```
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,Wo,Wc)
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,0.5)
[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,-0.5)
```

Description

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,Wo,Wc)` returns the numerator and denominator vectors, `Num` and `Den` respectively, of the target filter transformed from the real lowpass prototype by applying a first-order complex frequency shift transformation. This transformation rotates all the features of an original filter by the same amount specified by the location of the selected feature of the prototype filter, originally at W_o, placed at W_t in the target filter.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with the numerator specified by `B` and the denominator specified by `A`.

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,0.5)` calculates the allpass filter for doing the Hilbert transformation, i.e. a 90 degree counterclockwise rotation of an original filter in the frequency domain.

`[Num,Den,AllpassNum,AllpassDen] = iirshiftc(B,A,0,-0.5)` calculates the allpass filter for doing an inverse Hilbert transformation, i.e. a 90 degree clockwise rotation of an original filter in the frequency domain.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

Rotate all features of the prototype filter in the frequency domain by the same amount by specifying where the selected feature of an original filter, W_o= 0.5, should appear in the target filter, W_t= 0.25:

```
[num, den] = iirshiftc(b, a, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, num, den);
```

After applying the shift, the selected feature from the original filter is just where it should be, at W_t = 0.25.

Arguments

Variable	Description
B	Numerator of the prototype lowpass filter
A	Denominator of the prototype lowpass filter

iirshiftc

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Num	Numerator of the target filter
Den	Denominator of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

References   Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On digital differentiators, Hilbert transformers, and half-band low-pass filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

See Also   iirftransf | allpassshiftc | zpkshiftc

Purpose   Filter impulse response

Syntax   ```
[h,t] = impz(hfilt)
[h,t] = impz(hfilt,n)
[h,t] = impz(hfilt,n,fs)
[h,t] = impz(hfilt,[],fs)
impz(hfilt)
[h,t] = impz(hs)
[h,t] = impz(hs,Name,Value)
impz(hs)
```

Description   impz returns the impulse response based on the current filter
                   coefficients. This section describes common impz operation with
                   adaptive filters, discrete-time filters, multirate filters, and filter System
                   objects. For more input options, refer to impz in Signal Processing
                   Toolbox documentation.

                   [h,t] = impz(hfilt) returns the impulse response h and the
                   corresponding time points w at which the impulse response of hfilt
                   is computed. The impulse response is evaluated at 10 1-second
                   intervals—(0:9)'.

                   [h,t] = impz(hfilt,n) returns the impulse response evaluated at
                   floor(n) 1-second intervals—(0:floor(n)-1)'.

                   [h,t] = impz(hfilt,n,fs) returns the impulse response evaluated at
                   floor(n) 1/fs-second intervals—(0:floor(n)-1)'/fs.

                   [h,t] = impz(hfilt,[],fs) returns the impulse response evaluated
                   at 10 1/fs-second intervals—(0:9)'/fs.

                   impz(hfilt) uses FVTool to plot the impulse response of the filter. You
                   can also provide the optional input arguments n and fs with this syntax.

                   [h,t] = impz(hs) returns the impulse response for the filter
                   System object hs. The impulse response is evaluated at 10 1-second
                   intervals—(0:9)'. You can also provide the optional input arguments
                   n and fs with this syntax.

impz

`[h,t] = impz(hs,Name,Value)` returns an impulse response with additional options specified by one or more `Name,Value` pair arguments.

`impz(hs)` uses FVTool to plot the impulse response of the filter System object `hs`.

Note You can use `impz` for both real and complex filters. When you omit the output arguments, `impz` plots only the real part of the impulse response.

Input Arguments

hfilt

`hfilt` is either:

- An adaptive `adaptfilt`, discrete-time `dfilt`, or multirate `mfilt` filter object
- A vector of adaptive, discrete-time, or multirate filter objects

The multirate filter impulse response is computed relative to the rate at which the filter is running. When you specify `fs` (the sampling rate) as an input argument, `impz` assumes the filter is running at that rate.

For multistage cascades, `impz` forms a single-stage multirate filter that is equivalent to the cascade. It then computes the response relative to the rate at which the equivalent filter is running. `impz` does not support all multistage cascades. The function analyzes only those cascades for which there exists an equivalent single-stage filter.

As an example, consider a 2-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an interpolation factor of 4. In this case, an equivalent single-stage filter exists with an overall interpolation factor of 8. `impz` uses this equivalent filter for the analysis. If a sampling frequency `fs` is specified as an input argument to `impz`, the function interprets `fs` as the rate at which the equivalent filter is running.

hs

Filter System object. The function supports the following filter structures.

- `dsp.FIRFilter`
- `dsp.BiquadFilter`
- `dsp.FIRInterpolator`
- `dsp.CICInterpolator`
- `dsp.FIRDecimator`
- `dsp.CICDecimator`
- `dsp.FIRRateConverter`

n

Number of samples.

> **Default:** 10

fs

Sampling frequency.

> **Default:** 1

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Arithmetic

```
`double' | 'single' | 'fixed'
```

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

System Object State	Coefficient Data Type	Rule
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

h

Complex, n-element impulse response vector. If hfilt is a vector of filters, h is a complex, length(hfilt)-by-n matrix of impulse response vectors corresponding to each filter in hfilt. If n is not specified, the function uses a default value of 8192.

For adaptive filters, h is the instantaneous impulse response.

t

Time vector of length n, in seconds. t consists of n points equally spaced from 0 to floor(n)/fs. If n is not specified, the function uses a default value of 10. If fs is not specified, the function uses a default value of 1.

Examples   Create a discrete-time filter for a fourth-order, lowpass elliptic filter with a cutoff frequency of 0.4 times the Nyquist frequency. Use a second-order sections structure to resist quantization errors. Plot the first 50 samples of the impulse response, along with the reference impulse response.

```
d = fdesign.lowpass(.4,.5,1,80);
% Create a design object for the prototype filter.
```

Use ellip to design a minimum order discrete-time filter in second-order section form.

```
hd=design(d,'ellip');
```

Convert hd to fixed-point, and plot the impulse response:

```
impz(hd);
axis([1 75 -0.2 0.35])
```
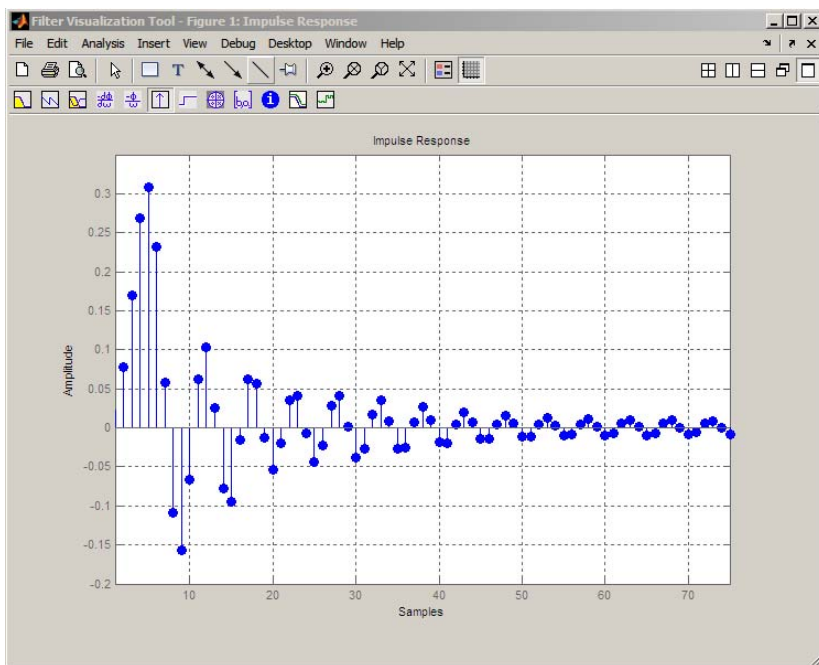
See Also   filter | stepz

info

Purpose	Information about filter

Syntax

```
info(h)
info(h,'short')
info(h,'long')
s = info(h)
```

Description   info(h) returns very basic information about the filter. The particulars depend on the filter type and structure.

info(h,'short') returns the same information as info(h).

info(h,'long') returns the following information about the filter:

- Specifications such as the filter structure and filter order

- Information about the design method and options

- Performance measurements for the filter response, such as the passband cutoff or stopband attenuation, included in the measure method

- Cost of implementing the filter in terms of operations required to apply the filter to data, included in the cost method

When the filter uses fixed-point arithmetic, the function returns additional information about the filter, including the arithmetic setting and details about the filter internals.

s = info(h) returns filter information in the variable s. You can also provide the optional arguments 'short' and 'long' with this syntax.

Input Arguments

h

One of the following types of filter object or filter System object:

- Discrete-time filter (dfilt) object

- Multirate filter (mfilt) object

- Adaptive filter (adaptfilt) object

- Filter System object. The function supports the following structures:

- dsp.FIRFilter

- dsp.BiquadFilter

- dsp.FIRInterpolator

- dsp.CICInterpolator

- dsp.FIRDecimator

- dsp.CICDecimator

- dsp.FIRRateConverter

'short'

Input string that instructs the function to return basic information about the filter.

'long'

Input string that instructs the function to return in-depth information about the filter.

Output Arguments	**s**

Variable for storing filter information.

Examples   Obtain short-format and long-format information about a filter.

```
>> d = fdesign.lowpass;
>> f = design(d);
>> info(f)
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 43
Stable            : Yes
Linear Phase      : Yes (Type 1)

>> info (f)
```

```
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 43
Stable            : Yes
Linear Phase      : Yes (Type 1)

>> info(f,'long')
Discrete-Time FIR Filter (real)
-------------------------------
Filter Structure  : Direct-Form FIR
Filter Length     : 43
Stable            : Yes
Linear Phase      : Yes (Type 1)

Design Method Information
Design Algorithm : equiripple

Design Options
DensityFactor : 16
MinOrder      : any
MinPhase      : false
StopbandDecay : 0
StopbandShape : flat

Design Specifications
Sampling Frequency : N/A (normalized frequency)
Response           : Lowpass
Specification      : Fp,Fst,Ap,Ast
Passband Edge      : 0.45
Stopband Edge      : 0.55
Passband Ripple    : 1 dB
Stopband Atten.    : 60 dB

Measurements
Sampling Frequency : N/A (normalized frequency)
Passband Edge      : 0.45
```

```
3-dB Point          : 0.46956
6-dB Point          : 0.48313
Stopband Edge       : 0.55
Passband Ripple     : 0.8919 dB
Stopband Atten.     : 60.9681 dB
Transition Width    : 0.1

Implementation Cost
Number of Multipliers : 43
Number of Adders      : 42
Number of States      : 42
MultPerInputSample    : 43
AddPerInputSample     : 42
```

See Also   coeffs | isfir | isstable | islinphase | dfilt

int

Purpose   States from CIC filter

Syntax   `integerstates = int(hm.states)`

Description   `integerstates = int(hm.states)` returns the states of a CIC filter in matrix form, rather than as the native `filtstates` object. An important point about `int` is that it quantizes the state values to the smallest number of bits possible while maintaining the values accurately.

Examples   For many users, the states of multirate filters are most useful as a matrix, but the CIC filters store the states as objects. Here is how you get the states from you CIC filter as a matrix.

```
hm = mfilt.cicdecim;
hs = hm.states; % Returns a FILTSTATES.CIC object hs.
states = int(hs); % Convert object hs to a signed integer matrix.
```

After using int to convert the states object to a matrix, here is what you get.

Before converting:

```
hm.states

ans =

    Integrator: [2x1 States]
          Comb: [2x1 States]
```

After the conversion and assigning the states to `states`:

```
states

states =

        0          0
        0          0
```

See Also   filtstates.cic | mfilt.cicdecim | mfilt.cicinterp

isallpass

Purpose	Determine whether filter is allpass

Syntax	`isallpass(hd)`
	`isallpass(hd,tolerance)`
	`isallpass(hs,...)`
	`isallpass(hs,'Arithmetic',`*`arithtype`*`)`

Description   `isallpass(hd)` determines whether the filter object `hd` is an allpass filter, returning `1` if true and `0` if false.

`isallpass(hd,tolerance)` uses input argument `tolerance` to determine whether the numerator and denominator transfer functions for the filter are close enough in value to be considered equal, and thus allpass. The function returns `1` if true (the difference between the numbers is less than `tolerance`) and `0` if not.

Given an allpass filter with this transfer function

$$H(z) = \frac{\alpha_n + \ldots + \alpha_1 z^{-(n-1)} + z^{-n}}{1 + \alpha_1 z^{-1} + \ldots + \alpha_n z^{-n}}$$

if the numerator and denominator transfer functions are equal, the filter is allpass. The `tolerance` input argument lets you determine how closely the transfer functions have to match to be considered equal. This approach may be most helpful in fixed-point allpass filters.

Lattice coupled allpass filters always have allpass sections, this function always returns `1` for filters whose structure is `latticeca`.

`isallpass(hs,...)` determines whether the filter System object `hs` is an allpass filter, returning `1` if true and `0` if false.

`isallpass(hs,'Arithmetic',`*`arithtype`*`)` analyzes the filter System object `hs` based on the specified *arithtype*. *arithtype* can be `'double'`, `'single'`, or `'fixed'`. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'`, the arithmetic changes depending on the setting of

the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on

isallpass

System Object State	Coefficient Data Type	Rule
		the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples   Use `dfilt.allpass` to construct an allpass filter. Then, use `isallpass` to test whether the filter is allpass.

```
c=[.8,1.5,0.4, 0.7]; % Allpass coefficients.
hd=dfilt.allpass(c);
AllPass = isallpass(hd);
% Returns a 1 indicated hd is all pass.
```

See Also   isfir | islinphase | ismaxphase | isminphase | isreal | issos | isstable

Purpose   Determine whether filter is FIR

Syntax   isfir(h)
                isfir(hs)

Description isfir(h) determines whether filter h is an FIR filter, returning 1 when
                the filter is an FIR filter, and 0 when it is IIR. isfir applies to dfilt,
                mfilt, and adaptfilt objects.

                To determine whether h is an FIR filter, isfir(h) inspects filter h and
                determines whether the filter, in transfer function form, has a scalar
                denominator. If it does, it is an FIR filter.

                isfir(hs) determines whether the filter System object hs is an FIR
                filter, returning 1 if true and 0 if false.

Examples   d = fdesign.lowpass;
                h = design(d);
                isfir(h)
                ans =

                      1

                returns 1 for the status of filter h. The filter is an FIR structure with
                denominator reference coefficient equal to 1.

                For multirate filters, isfir works the same way.

                d = fdesign.interpolator(5); % Interpolate by 5.
                h = design(d); % Use the default design method.
                isfir(h)

                ans =

                      1

                Use isfir with adaptive filters as well.

isfir

See Also   isallpass | islinphase | ismaxphase | isminphase | isreal | issos | isstable

Purpose   Determine whether filter is linear phase

Syntax   islinphase(h)
                 islinphase(h,tolerance)
                 islinphase(hs,...)
                 islinphase(hs,'Arithmetic',*arithtype*)

Description islinphase(h) determines if the filter object h is linear phase, and
                 returns 1 if true and 0 if false. adapfilt, dfilt, and mfilt objects
                 work with islinphase.

                 islinphase(h,tolerance) uses input argument tolerance to
                 determine whether the filter coefficients are close enough in value to
                 be considered symmetric or antisymmetric, returning 1 if true (the
                 difference between the values is less than tolerance) and 0 if not.

                 The phase determination is based on the reference coefficients. A filter
                 has linear phase if it is FIR and its transfer function coefficients are
                 symmetric or antisymmetric. If it is IIR, has poles on or outside the
                 unit circle, and both numerator and denominator are symmetric or
                 antisymmetric, it is linear phase also.

                 islinphase(hs,...) determines whether the filter System object hs is
                 linear phase, returning 1 if true and 0 if false.

                 islinphase(hs,'Arithmetic',*arithtype*) analyzes the filter
                 System object hs based on the specified *arithtype*. *arithtype* can
                 be 'double', 'single', or 'fixed'. When you specify 'double' or
                 'single', the function performs double- or single-precision analysis.
                 When you specify 'fixed' , the arithmetic changes depending on the
                 setting of the CoefficientDataType property and whether the System
                 object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples   This IIR filter has linear phase.

```
d = fdesign.lowpass('n,fc',10,0.55);
h = design(d,'window');
islinphase(h)
```

Using the specification nb,na,fp,fst results in an IIR filter that is not linear phase in this design.

```
nb=15
na=10
d=fdesign.lowpass('nb,na,fp,fst',nb,na,0.45,0.55)
h=design(d) % Use the default design method iirlpnorm.
islinphase(h)
```

See Also   isallpass | isfir | ismaxphase | isminphase | isreal | issos | isstable

ismaxphase

Purpose	Determine whether filter is maximum phase

Syntax	`ismaxphase(h)`
	`ismaxphase(h,tolerance)`
	`ismaxphase(hs,...)`
	`ismaxphase(hs,'Arithmetic',`*`arithtype`*`)`

Description

`ismaxphase(h)` determines if the filter object h is maximum phase, and returns 1 if true and 0 if false. `adapfilt`, `dfilt`, and `mfilt` objects work with `ismaxphase`.

`ismaxphase(h,tolerance)` uses input argument `tolerance` to determine whether the zeros of the filter transfer function have values that are close enough to 1 to be considered 1 or greater. Such values are on or outside the unit circle. The function returns 1 if true (the difference between the coefficient value and 1 is less than `tolerance`) and 0 if not.

The phase determination is based on the reference coefficients. A filter is maximum phase when the zeros of its transfer function are on or outside the unit circle, or when the numerator is a scalar.

`ismaxphase(hs,...)` determines whether the filter System object hs is maximum phase, returning 1 if true and 0 if false.

`ismaxphase(hs,'Arithmetic',`*`arithtype`*`)` analyzes the filter System object hs based on the specified *arithtype*. *arithtype* can be `'double'`, `'single'`, or `'fixed'`. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

ismaxphase

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples   Use `ismaxphase` to determine whether the following filters are maximum phase. The first is a discrete-time `dfilt` object and the second an adaptive filter.

```
fp = 100;
fst= 120;
fs = 800;
ap = 1;
ast= 60;
d = fdesign.lowpass('fp,fst,ap,ast',fp,fst,ap,ast,fs);
h = design(d,'equiripple','maxphase',true);
ismaxphase(h)
```

To make this a minimum phase filter, use `fliplr` to change the coefficient order. Reordering the coefficients this way changes the phase from maximum to minimum.

```
h.numerator=fliplr(h.numerator);
isminphase(h)
```

returns 1, so this filter is minimum phase.

For the adaptive filter example, try the following code:

```
x  = randn(1,500);     % Input to the filter
b  = fir1(31,0.5);     % FIR system to be identified
n  = 0.1*randn(1,500); % Observation noise signal
d  = filter(b,1,x)+n;  % Desired signal
mu = 1;                % NLMS step size
offset = 50;           % NLMS offset
ha = adaptfilt.nlms(32,mu,1,offset);
[y,e] = filter(ha,x,d);
```

After adapting, ha is an FIR filter that does not exhibit maximum phase.

```
ismaxphase(ha)
```

See Also   isallpass | isfir | islinphase | isminphase | isreal | issos | isstable

isminphase

Purpose	Determine whether filter is minimum phase
Syntax	isminphase(h) isminphase(h,tolerance) isminphase(hs,...) isminphase(hs,'Arithmetic',*arithtype*)
Description	isminphase(h) determines if the filter object h is minimum phase, and returns 1 if true and 0 if false. isminphase works with adapfilt, dfilt, and mfilt objects.

isminphase(h,tolerance) uses the input argument tolerance to determine whether the filter transfer function zeros are outside the unit circle by more than tolerance. If the zeros do not lie outside the unit circle by more than tolerance, isminphase returns a 1. If any zeros are more than tolerance outside the unit circle, isminphase returns a 0.

A filter is *minimum phase* when all the zeros of its transfer function are on or inside the unit circle, or the numerator is a scalar. An equivalent definition for a minimum phase filter is a causal and stable system with a causal and stable inverse.

isminphase(hs,...) determines whether the filter System object hs is minimum phase, returning 1 if true and 0 if false.

isminphase(hs,'Arithmetic',*arithtype*) analyzes the filter System object hs based on the specified *arithtype*. *arithtype* can be 'double', 'single', or 'fixed'. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed' , the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

isminphase

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples   Create a minimum-phase filter.

```
fp = 200;
fst= 230;
fs = 900;
ap = 1;
ast= 60;
d = fdesign.lowpass('fp,fst,ap,ast',fp,fst,ap,ast,fs);
h = design(d,'equiripple','minphase',true);
isminphase(h)
% Returns a 1
fvtool(h,'analysis','polezero')
```
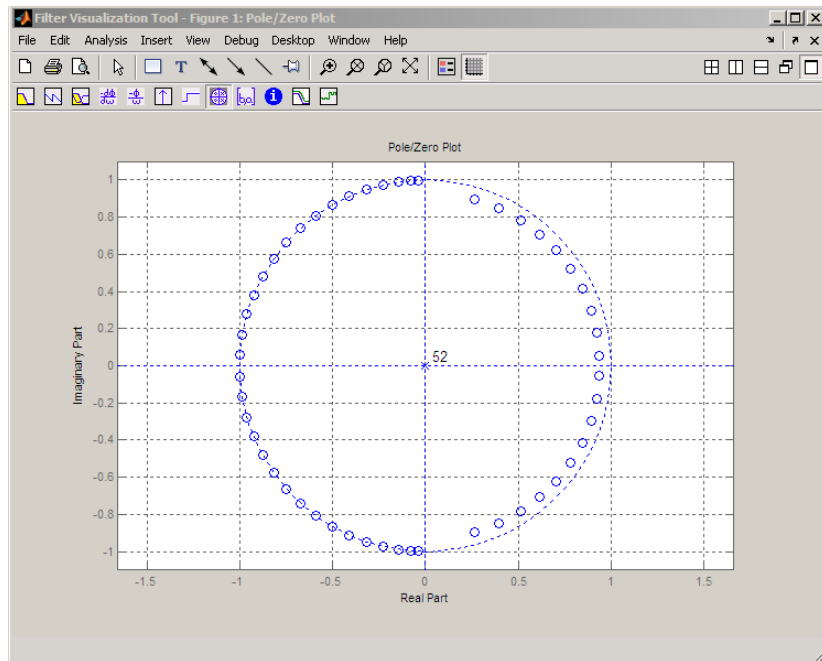
The zeros of the FIR filter are all on or inside the unit circle.

To illustrate the effect of coefficient quantization, make h a fixed-point filter. The quantization process results in the filter no longer being minimum phase. However, if the tolerance is increased to 0.01, the quantized design satisfies the definition of a minimum phase filter.

```
h.arithmetic='fixed';
isminphase(h)
% Returns a 0
fvtool(h,'analysis','polezero');
isminphase(h,0.01)
% Returns a 1
```

isminphase
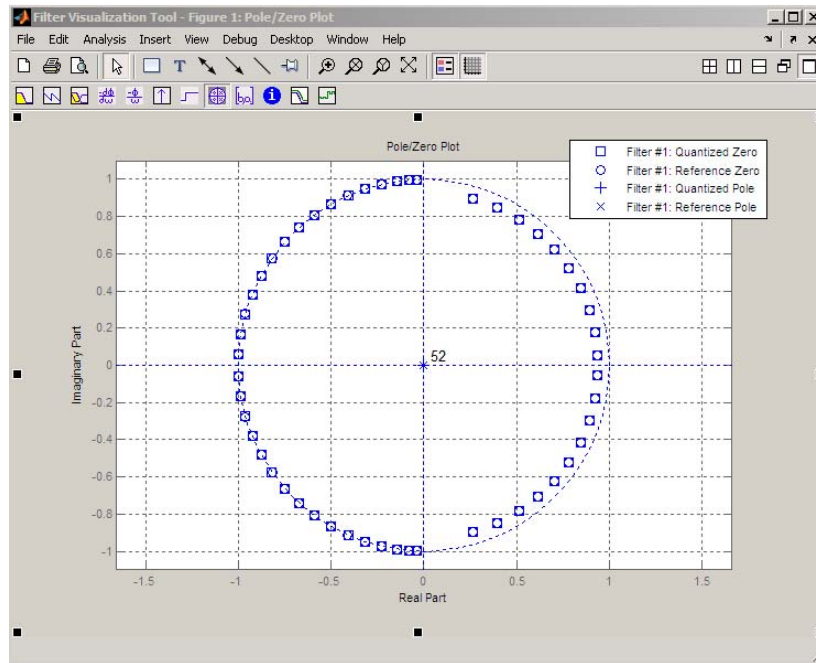
See Also   isallpass | isfir | islinphase | ismaxphase | isreal | issos |
isstable

isreal

Purpose   Determine whether filter uses real coefficients

Syntax   isreal(hd)
              isreal(hs)

Description   isreal(hd) returns 1 (or true) if all filter coefficients for the filter hd
                 are real, and returns 0 (or false) otherwise. Complex filters have one or
                 more coefficients with nonzero imaginary parts.

                 isreal(hs) determines whether the filter coefficients of the filter
                 System object hs are real, returning 1 if true and 0 if false.

                 Note Quantizing a filter cannot make a real filter into a complex filter.

Examples   Create a double-precision filter and a fixed-point filter. Then, test the
               coefficients of the fixed-point filter to see if they are strictly real.

               d=fdesign.lowpass('n,fp,ap,ast',5,0.4,0.5,20);
               hd=design(d,'ellip');
               hd.arithmetic='fixed';
               IsReal =isreal(hd);
               % Returns a 1

See Also   isfir | islinphase | ismaxphase | isminphase | issos | isstable |
               isallpass

issos

Purpose	Determine whether filter is SOS form
Syntax	`issos(hd)` `issoss(hs)`
Description	`issos(hd)` determines whether quantized filter `hq` consists of second-order sections. Returns `1` if all sections of quantized filter `hq` have order less than or equal to two, and `0` otherwise. `issoss(hs)` determines whether the filter System object `hs` consists of second-order sections, returning `1` if true and `0` if false.
Examples	By default, fdesign and design return SOS filters when possible. This example designs a lowpass SOS filter that uses fixed-point arithmetic. ```\nd=fdesign.lowpass('n,fp,ap,ast',40,0.55,0.1,60);\nhd=design(d,'ellip');\nhd.arithmetic='fixed';\nIsSOS=issos(hd);\n``` The fixed-point filter `hd` is in second-order section form, as is the double-precision version.
See Also	`isallpass` | `isfir` | `islinphase` | `ismaxphase` | `isminphase` | `isreal` | `isstable`

Purpose   Determine whether filter is stable

Syntax
```
isstable(hq)
isstable(hs)
isstable(hs,'Arithmetic',arithtype)
```

Description   isstable(hq) tests quantized filter hq to determine whether its poles are inside the unit circle. If the poles lie on or outside the circle, isstable returns 0. If the poles are inside the circle, isstable returns 1.

To determine the filter stability, isstable checks the filter coefficients. When the poles lie on or inside the unit circle, the filter is stable. FIR filters are stable by design because the defining transfer functions do not have denominator polynomials, thus no feedback to cause instability.

isstable(hs) determines whether the filter System object hs is stable, returning 1 if true and 0 if false.

isstable(hs,'Arithmetic',*arithtype*) analyzes the filter System object hs based on the specified *arithtype*. *arithtype* can be 'double', 'single', or 'fixed'. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed' , the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point

| System Object State	Coefficient Data Type	Rule
		analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples   Because filter stability is very important in your design process, use
isstable to determine whether your double-precision or fixed-point
IIR filter is stable.

```
d=fdesign.nyquist(2,'n,tw',24,0.1);
hd=design(d,'iirlinphase')
isstable(hd)
hd2=design(d,'equiripple');
isstable(hd2)
```

See Also   isallpass | isfir | islinphase | ismaxphase | isminphase |
isreal | issos | zplane

kaiserwin

Purpose	Kaiser window filter from specification object
Syntax	h = design(d,'kaiserwin') h = design(d,'kaiserwin',designoption,value,designoption,... value,...)

Description   h = design(d,'kaiserwin') designs a digital filter hd, or a multirate filter hm that uses a Kaiser window. For kaiserwin to work properly, the filter order in the specifications object must be even. In addition, higher order filters (filter order greater than 120) tend to be more accurate for smaller transition widths. kaiserwin returns a warning when your filter order may be too low to design your filter accurately.

h =
design(d,'kaiserwin',designoption,value,designoption,...
value,...) returns a filter where you specify design options as input arguments and the design process uses the Kaiser window technique.

To determine the available design options, use designopts with the specification object and the design method as input arguments as shown.

```
designopts(d,'method')
```

For complete help about using kaiserwin, refer to the command line help system. For example, to get specific information about using kaiserwin with d, the specification object, enter the following at the MATLAB prompt.

```
help(d,'kaiserwin')
```

Examples   This example designs a direct form FIR filter from a halfband filter specification object.

```
d=fdesign.halfband('n,tw',200,0.01)
hd= design(d,'kaiserwin','filterstructure','dffir')
```

In this example, kaiserwin uses an interpolating filter specification object:

```
d=fdesign.interpolator(4,'lowpass');
h = design(d,'kaiserwin');
```

See Also   equiripple | firls

lagrange

Purpose	Fractional delay filter from `fdesign.fracdelay` specification object
Syntax	`Hd = design(d,'lagrange')` `hd = design(d,'lagrange',FilterStructure,`*`structure`*`)`
Description	`Hd = design(d,'lagrange')` designs a fractional delay filter using the Lagrange method based on the specifications in `d`. `hd = design(d,'lagrange',FilterStructure,`*`structure`*`)` specifies the Lagrange design method and the *structure* filter structure for `hd`. The sole valid filter structure string for `structure` is `fd`, describing the fractional delay structure.
Examples	This example uses a fractional delay of 0.30 samples. The `help` and `designopts` commands provide the details about designing fractional delay filters.

```
d=fdesign.fracdelay(.30)

d =

                Response: 'Fractional Delay'
           Specification: 'N'
             Description: {'Filter Order'}
               FracDelay: 0.3
       NormalizedFrequency: true
             FilterOrder: 3

designmethods(d)


Design Methods for class fdesign.fracdelay (N):

lagrange

help(d,'lagrange')
```

```
 DESIGN Design a Lagrange fractional delay filter.
 HD = DESIGN(D, 'lagrange') designs a Lagrange filter
specified by the FDESIGN object D.

HD = DESIGN(..., 'FilterStructure', STRUCTURE) returns
a filter with the structure STRUCTURE.  STRUCTURE is 'fd'
by default and can be any of the following:

    'fd'

% Example #1 - Design a linear Lagrange fractional
% delay filter of 0.2 samples.
h = fdesign.fracdelay(0.2,'N',2);
Hd = design(h, 'lagrange', 'FilterStructure', 'fd')

% Example #2 - Design a cubic Lagrange fractional
% delay filter.
Fs = 8000;      % Sampling frequency of 8kHz
fdelay = 50e-6; % Fractional delay of 50 microseconds.
h  = fdesign.fracdelay(fdelay,'N',3,Fs);
Hd = design(h, 'lagrange', 'FilterStructure', 'fd');
```

This example designs a linear Lagrange fractional delay filter where
you set the delay to 0.2 seconds and the filter order N to 2.

```
h = fdesign.fracdelay(0.2,'N',2); .
hd = design(h,'lagrange','FilterStructure','fd')
```

Design a cubic Lagrange fractional delay filter with filter order equal
to 3..

```
Fs = 8000;      % Sampling frequency of 8 kHz.
fdelay = 50e-6; % Fractional delay of 50 microseconds.
h  = fdesign.fracdelay(fdelay,'N',3,Fs);
hd = design(h,'lagrange','FilterStructure','fd');
```

lagrange

References   Laakso, T. I., V. Välimäki, M. Karjalainen, and Unto K. Laine, "Splitting the Unit Delay - Tools for Fractional Delay Filter Design," *IEEE Signal Processing Magazine*, Vol. 13, No. 1, pp. 30-60, January 1996.

See Also   design | designmethods | designopts | fdesign | fdesign.fracdelay

Purpose   Check model for blocks from specific DSP System Toolbox libraries

Syntax   liblinks(lib)
                   liblinks(lib,sys)
                   liblinks(lib,sys,c)

Description   liblinks(lib) returns a cell array of strings that lists the blocks in the
                   current model that are linked to the specified libraries. The input lib
                   provides a cell array of strings with the library names. Use the library
                   name visible in the title bar when you open a library model.

                   liblinks(lib,sys) acts on the named model sys.

                   liblinks(lib,sys,c) changes the foreground color of the returned
                   blocks to the color c. Possible values of c are 'blue', 'green', 'red',
                   'cyan', 'magenta', 'yellow', or 'black'.

Examples   Check for blocks from the Sources library in the specified model:

                   rlsdemo
                   liblinks('dspsrcs4',gcs)

See Also   dsp_links

limitcycle

Purpose   Response of single-rate, fixed-point IIR filter

Syntax   report = limitcycle(hd)
                 report = limitcycle(hd,ntrials,inputlengthfactor,stopcriterion)

Description report = limitcycle(hd) returns the structure report that contains
                 information about how filter hd responds to a zero-valued input vector.
                 By default, the input vector has length equal to twice the impulse
                 response length of the filter.

                 limitcycle returns a structure whose elements contain the details
                 about the limit cycle testing. As shown in this table, the report includes
                 the following details.

Output Object Property	Description
LimitCycleType	Contains one of the following results: • Granular — indicates that a granular overflow occurred. • Overflow — indicates that an overflow limit cycle occurred. • None — indicates that the test did not find any limit cycles.
Zi	Contains the initial condition value(s) that caused the detected limit cycle to occur.
Output	Contains the output of the filter in the steady state.
Trial	Returns the number of the Monte Carlo trial on which the limit cycle testing stopped. For example, Trial = 10 indicates that testing stopped on the tenth Monte Carlo trial.

Using an input vector longer than the filter impulse response ensures that the filter is in steady-state operation during the limit cycle testing. `limitcycle` ignores output that occurs before the filter reaches the steady state. For example, if the filter impulse length is 500 samples, `limitcycle` ignores the filter output from the first 500 input samples.

To perform limit cycle testing on your IIR filter, you must set the filter `Arithmetic` property to `fixed` and `hd` must be a fixed-point IIR filter of one of the following forms:

- `df1` — direct-form I
- `df1t` — direct-form I transposed
- `df1sos` — direct-form I with second-order sections
- `df1tsos` — direct-form I transposed with second-order sections
- `df2` — direct-form II
- `df2t` — direct-form II transposed
- `df2sos` — direct-form II with second-order sections
- `df2tsos` — direct-form II transposed with second-order sections

When you use `limitcycle` without optional input arguments, the default settings are

- Run 20 Monte Carlo trials
- Use an input vector twice the length of the filter impulse response
- Stop testing if the simulation process encounters either a granular or overflow limit cycle

To determine the length of the filter impulse response, use `impzlength`:

```
impzlength(hd)
```

During limit cycle testing, if the simulation runs reveal both overflow and granular limit cycles, the overflow limit cycle takes precedence and is the limit cycle that appears in the report.

Each time you run `limitcycle`, it uses a different sequence of random initial conditions, so the results can differ from run to run.

Each Monte Carlo trial uses a new set of randomly determined initial states for the filter. Test processing stops when `limitcycle` detects a zero-input limit cycle in filter `hd`. `report = limitcycle(hd,ntrials,inputlengthfactor,stopcriterion)` lets you set the following optional input arguments:

- `ntrials` — Number of Monte Carlo trials (default is 20).

- `inputlengthfactor` — integer factor used to calculate the length of the input vector. The length of the input vector comes from (`impzlength(hd) * inputlengthfactor`), where `inputlengthfactor = 2` is the default value.

- `stopcriterion` — the criterion for stopping the Monte Carlo trial processing. `stopcriterion` can be set to **either** (the default), **granular**, **overflow**. This table describes the results of each stop criterion.

stopcriterion Setting	Description
either	Stop the Monte Carlo trials when `limitcycle` detects either a granular or overflow limit cycle.
granular	Stop the Monte Carlo trials when `limitcycle` detects a granular limit cycle.
overflow	Stop the Monte Carlo trials when `limitcycle` detects an overflow limit cycle.

> **Note** An important feature is that if you specify a specific limit cycle stop criterion, such as overflow, the Monte Carlo trials do not stop when testing encounters a granular limit cycle. You receive a warning that no overflow limit cycle occurred, but consider that a granular limit cycle might have occurred.

Examples   In this example, there is a region of initial conditions in which no limit cycles occur and a region where they do. If no limit cycles are detected before the Monte Carlo trials are over, the state sequence converges to zero. When a limit cycle is found, the states do not end at zero. Each time you run this example, it uses a different sequence of random initial conditions, so the plot you get can differ from the one displayed in the following figure.

```
s = [1 0 0 1 0.9606 0.9849];
hd = dfilt.df2sos(s);
hd.arithmetic = 'fixed';
greport = limitcycle(hd,20,2,'granular')
oreport = limitcycle(hd,20,2,'overflow')
figure,
subplot(211),plot(greport.Output(1:20)), title('Granular Limit Cycle');
subplot(212),plot(oreport.Output(1:20)), title('Overflow Limit Cycle');

greport =

    LimitCycle: 'granular'
            Zi: [2x1 double]
        Output: [1303x1 embedded.fi]
         Trial: 1


oreport =

    LimitCycle: 'overflow'
            Zi: [2x1 double]
        Output: [1303x1 embedded.fi]
```

```
Trial: 2
```

The plots shown in this figure present both limit cycle types — the first displays the small amplitude granular limit cycle, the second the larger amplitude overflow limit cycle.
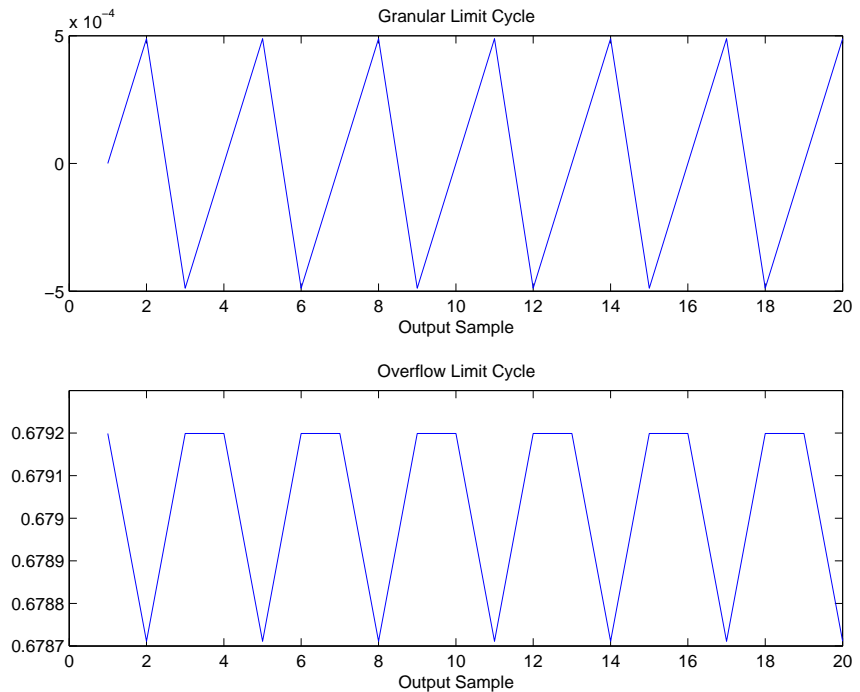
As you see from the plots, and as is generally true, overflow limit cycles are much greater magnitude than granular limit cycles. This is why `limitcycle` favors overflow limit cycle detection and reporting.

See Also   `freqz | noisepsd`

Purpose   Maxflat FIR filter

Syntax   hd = design(d,'maxflat')
                 hd = design(d,'maxflat','FilterStructure',structure)

Description hd = design(d,'maxflat') designs a maximally flat filter, hd, from a filter specification object, d.

hd = design(d,'maxflat','FilterStructure',structure) designs a maximally flat filter where structure is one of the following:

- 'dffir', a discrete-time, direct-form FIR filter (the default value)

- 'dffirt', a discrete-time, direct-form FIR transposed filter

- 'dfsymfir', a discrete-time, direct-form symmetric FIR filter

- 'fftfir', a discrete-time, overlap-add, FIR filter

Examples   Example 1: Design a lowpass filter with a maximally flat FIR structure.

```
d  = fdesign.lowpass('N,F3dB', 50, 0.3);
Hd = design(d, 'maxflat');
```

Example 2: Design a highpass filter with a maximally flat overlap-add FIR structure.

```
d  = fdesign.highpass('N,F3dB', 50, 0.7);
Hd=design(d,'maxflat','FilterStructure','fftfir');
```

maximizestopband

Purpose	Maximize stopband attenuation of fixed-point filter
Syntax	`Hq = maximizestopband(Hd,Wordlength)` `Hq = maximizestopband(Hd,Wordlength,'Ntrials',N)`
Description	`Hq = maximizestopband(Hd,Wordlength)` quantizes the single-stage or multistage FIR filter `Hd` and returns the fixed-point filter `Hq` with wordlength `wordlength` that maximizes the stopband attenuation. `Hd` must be generated using `fdesign` and `design`. For multistage filters, `wordlength` can either be a scalar or vector. If `wordlength` is a scalar, the same wordlength is used for all stages. If `wordlength` is a vector, each stage uses the corresponding element in the vector. The vector length must equal the number of stages. `maximizestopband` uses a stochastic noise-shaping procedure by default to minimize the wordlength. To obtain repeatable results on successive function calls, initialize the uniform random number generator `rand`
	`Hq = maximizestopband(Hd,Wordlength,'Ntrials',N)` specifies the number of Monte Carlo trials to use in the maximization. `Hq` is the fixed-point filter with the largest stopband attenuation among the trials. The number of Monte Carlo trials `N` defaults to 1.
	You must have the Fixed-Point Toolbox software installed to use this function.
Examples	Maximize stopband attenuation for 16–bit fixed-point filter.

```
Hf = fdesign.lowpass('Fp,Fst,Ap,Ast',0.4,0.45,0.5,60);
Hd = design(Hf,'equiripple');
WL = 16; % use 16 bits to represent coefficients
Hq = maximizestopband(Hd,WL);
% Compare stopband attenuation
md = measure(Hd);
mq = measure(Hq);
hfvt=fvtool(Hd,Hq,'showreference','off');
legend(hfvt,'Floating-point Filter','Fixed-point Filter');
```
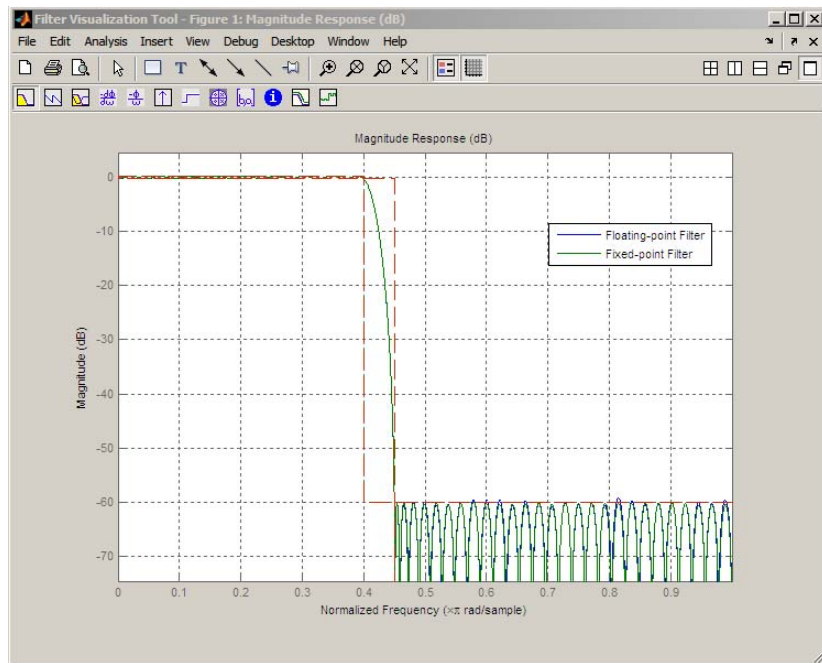
See Also   constraincoeffwl | design | fdesign | minimizecoeffwl | measure
| rand

Tutorials   • "Fixed-Point Data Types"

maxstep

Purpose   Maximum step size for adaptive filter convergence

Syntax   mumax = maxstep(ha,x)
                 [mumax,mumaxmse] = maxstep(ha,x)

Description mumax = maxstep(ha,x) predicts a bound on the step size to provide
                 convergence of the mean values of the adaptive filter coefficients. The
                 columns of the matrix x contain individual input signal sequences. The
                 signal set is assumed to have zero mean or nearly so.

                 [mumax,mumaxmse] = maxstep(ha,x) predicts a bound on the adaptive
                 filter step size to provide convergence of the LMS adaptive filter
                 coefficients in the mean-square sense. maxstep issues a warning when
                 ha.stepsize is outside of the range 0 < ha.stepsize < mumaxmse/2.

                 maxstep is available for the following adaptive filter objects:

                 • adaptfilt.blms

                 • adaptfilt.blmsfft

                 • adaptfilt.lms

                 • adaptfilt.nlms (uses a different syntax. Refer to the text below.)

                 • adaptfilt.se

                 Note With adaptfilt.nlms filter objects, maxstep uses the following
                 slightly different syntax:

                 mumax = maxstep(ha)
                 [mumax,mumaxmse] = maxstep(ha)

                 The maximum step size for convergence is fully defined by the filter
                 object ha. Matrix x is not necessary. If you include an x input matrix,
                 MATLAB returns an error.

Examples   Analyze and simulate a 32-coefficient (31st-order) LMS adaptive filter
object. To demonstrate the adaptation process, run 2000 iterations
and 50 trials.

```
% Specify [numiterations,numexamples] = size(x);
x = zeros(2000,50);
d = x;
obj = fdesign.lowpass('n,fc',31,0.5);
hd = design(obj,'window'); % FIR filter to identified.
coef = cell2mat(hd.coefficients); % Convert cell array to matrix.

for k=1:size(x,2);   % Create input and desired response signal
                        % matrices.
% Set the (k)th input to the filter.
  x(:,k) = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x,1),1)));
  n = 0.1*randn(size(x,1),1);  % (k)th observation noise signal.
  d(:,k) = filter(coef,1,x(:,k))+n; % (k)th desired signal end.
end
mu = 0.1;                       % LMS step size.
ha = adaptfilt.lms(32,mu);
[mumax,mumaxmse] = maxstep(ha,x);

Warning: Step size is not in the range 0 < mu < mumaxmse/2:
Erratic behavior might result.

mumax

mumax =

    0.0623

mumaxmse

mumaxmse =

    0.0530
```

maxstep

See Also   msepred | msesim | filter

Purpose   Measure filter magnitude response

Syntax   measure(hd)
measure(hm)
measure(hs)
measure(hs)

Description   measure(hd) returns measured values for specific points in the
magnitude response curve for filter object hd. When you use a design
object d to create a filter (by using fdesign.*type* to create d), you
specify one or more values that define your desired filter response.
measure(hd) tests the filter to determine the actual values in the
magnitude response of the filter, such as the stopband attenuation or
the passband ripple. Comparing the results returned by measure to
the specifications you provided in the design object helps you assess
whether the filter meets your design criteria.

Note To use measure, hd, hm, or hs must result from using a filter
design method with a filter specifications object. measure works with
multirate filters and discrete-time filters. The function does not support
adaptive filters because you cannot use fdesign.*type* to construct
adaptive filter specifications objects.

---

measure(hd) returns measurements for the discrete-time filter hd. The
function determines the relevant specifications based on the response
type of the design object you use to create the filter. For example, for
single-rate lowpass filters made from design objects, measure(hd)
returns the following filter specifications.

Lowpass Filter Specification	Description
Sampling Frequency	Filter sampling frequency.
Passband Edge	Location of the edge of the passband as it enters transition.

Lowpass Filter Specification	Description
3-dB Point	Location of the –3 dB point on the response curve.
6-dB Point	Location of the –6 dB point on the response curve.
Stopband Edge	Location of the edge of the transition band as it enters the stopband.
Passband Ripple	Ripple in the passband.
Stopband Atten.	Attenuation in the stopband.
Transition Width	Width of the transition between the passband and stopband, in normalized frequency or absolute frequency. Measured between Fpass and Fstop.

In contrast, when you use a bandstop design object, measure(hd) returns these specifications for the resulting bandstop filter.

Bandstop Filter Specification	Description
Sampling Frequency	Filter sampling frequency.
First Passband Edge	Location of the edge of the first passband.
First 3-dB Point	Location of the edge of the –3 dB point in the first transition band.
First 6-dB Point	Location of the edge of the –6 dB point in the first transition band.
First Stopband Edge	Location of the start of the stopband.
Second Stopband Edge	Location of the end of the stopband.

Bandstop Filter Specification	Description
Second 6-dB Point	Location of the edge of the −6 dB point in the second transition band.
Second 3-dB Point	Location of the edge of the −3 dB point in the second transition band.
Second Passband Edge	Location of the start of the second passband.
First Passband Ripple	Ripple in the first passband.
Stopband Atten.	Attenuation in the stopband.
Second Passband Edge	Ripple in the second passband.
First Transition Width	Width of the first transition region. Measured between the −3 and −6 dB points.
Second Transition Width	Width of the second transition region. Measured between the −6 and −3 dB points.

measure(hm) returns measurements for the multirate filter hm. The set of filter specifications that measure returns for multirate filters might be different from those for discrete-time filters.

The set of response measurements that measure returns depends on the response you use to design the filter. For example, when hm is an FIR lowpass interpolator (whose response is lowpass), measure(hm) returns the following set of measurements.

Interpolator Filter Specification	Description
First Passband Edge	Location of the edge of the passband as it enters transition.
3-dB Point	Location of the −3 dB point on the response curve.

Interpolator Filter Specification	Description
`6-dB Point`	Location of the –6 dB point on the response curve.
`Stopband Edge`	Location of the edge of the transition band as it enters the stopband.
`Passband Ripple`	Ripple in the passband.
`Stopband Atten.`	Attenuation in the stopband.
`Transition Width`	Width of the transition between the passband and stopband, in normalized frequency or absolute frequency. Measured between `Fpass` and `Fstop`.

For reference, the specification object d created the interpolator specifications shown in the preceding table:

```
d=fdesign.interpolator(6,'lowpass')

d =

        MultirateType: 'Interpolator'
   InterpolationFactor: 6
             Response: 'Lowpass'
        Specification: 'Fp,Fst,Ap,Ast'
          Description: {4x1 cell}
   NormalizedFrequency: true
                Fpass: 0.133333333333333
                Fstop: 0.166666666666667
                Apass: 1
                Astop: 60
```

measure(hs) returns measurements for the filter System object hs. You must construct hs using design with the optional name-value pair argument 'SystemObject',true.

measure(hs) returns measurements for the filter System object hs with additional options specified by one or more Name,Value pair arguments.

Tips   For designs that do not specify some of the frequency constraints, the function may not be able to determine corresponding magnitude measurements. In these cases, a constraint can be passed in to measure to determine such measurements. For example:

```
f = fdesign.lowpass('N,F3dB,Ast',8,0.5,80);
H = design(f,'cheby2','SystemObject',true);
measure(H)
```

returns values of Unknown for the passband edge, passband ripple, and transition width measurements, but

```
f = fdesign.lowpass('N,F3dB,Ast',8,0.5,80);
H = design(f,'cheby2','SystemObject',true);
measure(H,'Fpass',0.4)
```

provides measurements for all returned values.

Input Arguments

hd

Discrete-time dfilt filter object.

hm

Multirate mfilt filter object.

hs

Filter System object. The function supports the following filter structures.

- dsp.FIRFilter

- dsp.BiquadFilter

- dsp.FIRInterpolator

- dsp.CICInterpolator

- dsp.FIRDecimator

- dsp.CICDecimator

- dsp.FIRRateConverter

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Arithmetic

`double' | 'single' | 'fixed'

For filter System object inputs only, specify the arithmetic used during analysis. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed', the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point

System Object State	Coefficient Data Type	Rule
		analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Examples   For the first example, create a lowpass filter and check whether the
actual filter meets the specifications. For this case, use normalized
frequency for Fs, the default setting.

```
d2=fdesign.lowpass('Fp,Fst,Ap,Ast',0.45,0.55,0.1,80)

d2 =

                   Response: 'Lowpass'
              Specification: 'Fp,Fst,Ap,Ast'
                Description: {4x1 cell}
        NormalizedFrequency: true
                      Fpass: 0.45
                      Fstop: 0.55
                      Apass: 0.1
                      Astop: 80

designmethods(d2)


Design Methods for class fdesign.lowpass (Fp,Fst,Ap,Ast):


butter
cheby1
cheby2
ellip
equiripple
ifir
kaiserwin
multistage

hd2=design(d2) % Use the default equiripple design method.

hd2 =

      FilterStructure: 'Direct-Form FIR'
```

```
         Arithmetic: 'double'
          Numerator: [1x68 double]
    PersistentMemory: false

measure(hd2)

ans =

Sampling Frequency : N/A (normalized frequency)
Passband Edge      : 0.45
3-dB Point         : 0.47794
6-dB Point         : 0.48909
Stopband Edge      : 0.55
Passband Ripple    : 0.09615 dB
Stopband Atten.    : 80.2907 dB
Transition Width   : 0.1
```

`Stopband Edge`, `Passband Edge`, `Passband Ripple`, and `Stopband Atten.` all meet the specifications.

Now, using `Fs` in linear frequency, create a bandpass filter, and measure the magnitude response characteristics.

```
d=fdesign.bandpass

d =

               Response: 'Bandpass'
          Specification: 'Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2'
            Description: {7x1 cell}
    NormalizedFrequency: true
                 Fstop1: 0.35
                 Fpass1: 0.45
                 Fpass2: 0.55
                 Fstop2: 0.65
                 Astop1: 60
                  Apass: 1
                 Astop2: 60
```

```
normalizefreq(d,false,1.5e3) % Convert to linear freq.

hd=design(d,'cheby2');

measure(hd)

ans =

Sampling Frequency      : 1.5 kHz
First Stopband Edge     : 0.2625 kHz
First 6-dB Point        : 0.31996 kHz
First 3-dB Point        : 0.32497 kHz
First Passband Edge     : 0.3375 kHz
Second Passband Edge    : 0.4125 kHz
Second 3-dB Point       : 0.42503 kHz
Second 6-dB Point       : 0.43004 kHz
Second Stopband Edge    : 0.4875 kHz
First Stopband Atten.   : 60 dB
Passband Ripple         : 0.17985 dB
Second Stopband Atten.  : 60 dB
First Transition Width  : 0.075 kHz
Second Transition Width : 0.075 kHz
```

measure(hd) returns the actual response values, in the units you
chose. In this example, all frequencies appear in kilohertz because the
sampling frequency is in kilohertz.

See Also   design | fdesign | normalizefreq

Purpose   Multirate filter

Syntax   hm = mfilt.*structure*(input1,input2,...)

Description hm = mfilt.*structure*(input1,input2,...) returns the object hm
                 of type *structure*. As with dfilt and adaptfilt objects, you must
                 include the *structure* string to construct a multirate filter object.
                 You can, however, construct a default multirate filter object of a given
                 structure by not including input arguments in your calling syntax.

                 Multirate filters include decimators and interpolators, and fractional
                 decimators and fractional interpolators where the resulting
                 interpolation or decimation factor is not an integer.

Structures

Each of the following multirate filter structures has a reference page
of its own.

Filter Structure String	Description of Resulting Multirate Filter	Coefficient Mapping Support in `realizemdl` and `block`
mfilt.cascade	Cascade multirate filters to form another filter	Supported
mfilt.cicdecim	Cascaded integrator-comb decimator	Not supported
mfilt.cicinterp	Cascaded integrator-comb interpolator	Not supported
mfilt.farrowsrc	Multirate Farrow filter	Supported.

Filter Structure String	Description of Resulting Multirate Filter	Coefficient Mapping Support in `realizemdl` and `block`
`mfilt.fftfirinterp`	Overlap-add FIR polyphase interpolator	Not supported
`mfilt.firdecim`	Direct-form FIR polyphase decimator	Supported
`mfilt.firinterp`	Direct-form FIR polyphase interpolator	Supported
`mfilt.firsrc`	Direct-form FIR polyphase sample rate converter	Supported
`mfilt.firtdecim`	Direct-form transposed FIR polyphase decimator	Supported
`mfilt.holdinterp`	FIR hold interpolator	Not supported
`mfilt.iirdecim`	IIR decimator	Supported
`mfilt.iirinterp`	IIR interpolator	Supported
`mfilt.linearinterp`	FIR Linear interpolator	Supported
`mfilt.iirwdfdecim`	IIR wave digital filter decimator	Supported
`mfilt.iirwdfinterp`	IIR wave digital filter interpolator	Supported

Copying mfilt Objects

To create a copy of an `mfilt` object, use the copy method.

```
h2 = copy(hd)
```

Note The syntax hd2 = hd copies only the object handle. It does not create a new object. hd2 and hd are not independent. If you change the property value for one of the two, such as hd2, you are changing the property for both.

Examples Decimation by a factor of two. Convert input sampled at 48 kHz to 24 kHz:

```
Fs = 4.8e4;
t =0:1/Fs:1-(1/Fs);
x = cos(2*pi*4000*t);
Hm=mfilt.firdecim(2);
% Note cutoff frequency of 1/2 normalized frequency
fvtool(Hm);
% Note the group delay of 34 samples
fvtool(Hm,'analysis','grpdelay');
y = filter(Hm,x);
% Note delay in output is consistent with 36/2
stem(y(1:48),'markerfacecolor',[0 0 1]);
```

Using existing coefficients to decimate a signal by a factor of two:

```
M = 2; % Decimation factor
  b = firhalfband('minorder',.45,0.0001);
  Hm = mfilt.firdecim(M,b);
  % Decimate a signal which consists of the sum of 2 sinusoids.
  N = 160;
  x = sin(2*pi*.05*[0:N-1]+pi/3)+cos(2*pi*.03*[0:N-1]+pi/3);
  y = filter(Hm,x);
```

mfilt

> **Note** Multirate filters can also have complex coefficients. For example, you can specify complex coefficients in the argument num passed to the filter structure. This works for all multirate filter structures.
>
> ```
> m = 2;
> num = [0.5 0.5+1j*0.2];
> Hm = mfilt.firdecim(m, num);
> y = filter(Hm, [1:10]);
> ```

See Also   mfilt.firinterp | mfilt.firsrc | mfilt.firtdecim

Purpose   Cascade filter objects

Syntax   hm = cascade(hm1,hm2,...,hmn)

Description   hm = cascade(hm1,hm2,...,hmn) creates filter object hm by cascading
                 (connecting in series) the individual filter objects hm1, hm2, and so on
                 to hmn.

                 In block diagram form, the cascade looks like this, with x as the input to
                 the filter hm and y the output from the cascade filter hm:

                 mfilt.cascade accepts any combination of mfilt and dfilt objects
                 (discrete time filters) to cascade, as well as Farrow filter objects.

Examples   Create a variety of mfilt objects and cascade them together.

```
hm(1) = mfilt.firdecim(12);
hm(2) = mfilt.firdecim(4);
h1 = mfilt.cascade(hm(1),hm(2));
hm(3) = mfilt.firinterp(4);
hm(4) = mfilt.firinterp(12);
h2 = mfilt.cascade(hm(3),hm(4));
% Cascade h1 and h2 together
h3 = mfilt.cascade(h1,h2,9600);
```

See Also   dfilt.cascade

mfilt.cicdecim

Purpose	Fixed-point CIC decimator

Syntax   hm = mfilt.cicdecim(r,m,n,iwl,owl,wlps)

Description hm = mfilt.cicdecim(r,m,n,iwl,owl,wlps) returns a cascaded
integrator-comb (CIC) decimation filter object.

All of the input arguments are optional. To enter any optional value,
you must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies
the default values shown in the table below.

The following table describes the input arguments for creating hm.

Input Arguments	Description
r	Decimation factor applied to the input signal. Sharpens the response curve to let you change the shape of the response. r must be an integer value greater than or equal to 1. The default value is 2.
m	Differential delay. Changes the shape, number, and location of nulls in the filter response. Increasing m increases the sharpness of the nulls and the response between nulls. In practice, differential delay values of 1 or 2 are the most common. m must be an integer value greater than or equal to 1. The default value is 1.
n	Number of sections. Deepens the nulls in the response curve. Note that this is the number of either comb or integrator sections, not the total section count. 2 is the default value.
iwl	Word length of the input signal. Use any integer number of bits. The default value is 16 bits.

Input Arguments	Description
owl	Word length of the output signal. It can be any positive integer number of bits. By default, owl is 16 bits.
wlps	Defines the number of bits per word in each filter section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter wlps as a scalar or vector of length 2*n, where n is the number of sections. When wlps is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator.
	When you elect to specify wlps as an input argument, the FilterInternals property automatically switches from the default value of 'FullPrecision' to 'SpecifyWordLengths'.

Constraints and Word Length Considerations

CIC decimators have the following constraint — the word lengths of the filter section must be monotonically decreasing. The word length of each filter section must be the same size as, or smaller than, the word length of the previous filter section.

The formula for B_{max}, the most significant bit at the filter output, is given in the Hogenauer paper in the References below.

$$B_{\max} = (N \log_2 RM + B_{in} - 1)$$

where B_{in} is the number of bits of the input.

The cast operations shown in the diagram in "Algorithms" on page 7-1023 perform the changes between the word lengths of each section. When you specify word lengths that do not follow the constraints above, the constructor returns an error.

When you specify the word lengths correctly, the most significant bit B_{max} stays the same throughout the filter, while the word length of each section either decreases or stays the same. This can cause the fraction length to change throughout the filter as least significant bits are truncated to decrease the word length, as shown in "Algorithms" on page 7-1023.

Properties of the Object

Objects have properties that control the way the object behaves. This table lists all the properties for the filter, with a description of each.

Name	Values	Default	Description
Arithmetic	fixed	fixed	Reports the kind of arithmetic the filter uses. CIC decimators are always fixed-point filters.
DecimationFactor	Any positive integer	2	Amount to reduce the input sampling rate.
DifferentialDelay	Any positive integer	1	Sets the differential delay for the filter. Usually a value of one or two is appropriate.
FilterStructure	mfilt structure string	None	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt objects.
FilterInternals	FullPrecision, MinWordLengths, SpecifyPrecision, SpecifyWordLengths	FullPrecision	Set the usage mode for the filter. Refer to "Usage Modes" on page 7-1015 below for details.

Name	Values	Default	Description
InputFracLength	Any positive integer	15	The number of bits applied to the fraction length to interpret the input data to the filter.
InputOffset	Integers in the range $0 \leq \text{InputOffset} \leq r-1$	0	Contains a value derived from the number of input samples and the decimation factor — InputOffset = mod(length(nx),m) where nx is the number of input samples that have been processed so far and m is the decimation factor. The InputOffset property applies only when you set the PersistentMemory property to true. See "InputOffset" on page 8-125 for more information.
InputWordLength	Any positive integer	16	The number of bits applied to the word length to interpret the input data to the filter.

Name	Values	Default	Description
NumberOfSections	Any positive integer	2	Number of sections used in the decimator. Generally called n. Reflects either the number of decimator or comb sections, not the total number of sections in the filter.
OutputFracLength	Any positive integer	15	The number of bits applied to the fraction length to interpret the output data from the filter. Read-only.
OutputWordLength	Any positive integer	16	The number of bits applied to the word length to interpret the output data from the filter.
PersistentMemory	false or true	false	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States

Name	Values	Default	Description
			that the filter does not change are not affected. When `PersistentMemory` is `false`, you cannot access the filter states. Setting `PersistentMemory` to `true` reveals the `States` property so you can modify the filter states.

Name	Values	Default	Description
SectionWord Lengths	Any integer or a vector of length 2*n.	16	Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter `SectionWordLengths` as a scalar or vector of length 2*n, where n is the number of sections. When `SectionWordLengths` is a scalar, the scalar value is applied to each filter section. When `SectionWordLengths` is a vector of values,

Name	Values	Default	Description
			the values apply to the sections in order. The default is 16 for each section in the decimator. Available when `FilterInternals` is `'SpecifyWordLengths'`.
States	`filtstates.cic` object	m+1-by-n matrix of zeros, after you call function `int`.	Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. m is the differential delay of the comb section and n is the number of sections in the filter. The integrator states are stored in the first matrix row. States for the comb section fill the remaining rows in the matrix. Available for modification when `PersistentMemory` is `true`. Refer to the `filtstates` object in Signal Processing Toolbox documentation for more general information about the `filtstates` object.

Usage Modes

There are four modes of usage for this which are set using the `FilterInternals` property

- `FullPrecision` — All word and fraction lengths set to B_{max} + 1, called B_{accum} by Fred Harris in [3]. Full Precision is the default setting.

- `MinWordLengths` — Automatically set the sections for minimum word lengths.

- `SpecifyWordLengths` — Specify the word lengths for each section.

- `SpecifyPrecision` — Specify precision by providing values for the word and fraction lengths for each section.

Full Precision

In full precision mode, the word lengths of all sections and the output are set to B_{accum} as defined by

$$B_{accum} = ceil(N_{secs}(Log_2(D \times M)) + InputWordLength)$$

where N_{secs} is the number of filter sections.

Section fraction lengths and the fraction length of the output are set to the input fraction length.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'FullPrecision'
```

Minimum Wordlengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than B_{accum}, roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [1]) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than B_{accum} as well, so that the roundoff noise introduced by all sections does not exceed the roundoff noise introduced at the output.

In this mode, the design calculates the word lengths of each section to meet the Hogenauer criterion. The algorithm subtracts the number of bits computed using eq. 21 in Hogenauer's paper from B_{accum} to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore each bit thrown out at a particular section decrements the fraction length of that section by one bit compared to the input fraction length. Setting the output wordlength for the filter automatically sets the output fraction length as well.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'MinWordLengths'

OutputWordLength: 16
```

Specify word lengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length `2*(NumberOfSections)`, where each vector element represents the word length for a section. If you specify a scalar, such as B_{accum}, the full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicdecim;
hcic.FilterInternals='minwordlengths';
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display for the `SpecifyWordLengths` mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyWordLengths'

SectionWordLengths: [19 18 18 17]
```

```
OutputWordLength: 16
```

Specify precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the `SpecifyPrecision` mode, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the word length for each section. When you enter a scalar such as B_{accum}, `mfilt.cicdecim` expands that scalar to a vector of the appropriate size and applies the scalar value to each section and the output. The design does not check that this vector is monotonically decreasing.

Also, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the fraction length for each section as well. When you enter a scalar such as B_{accum}, `mfilt.cicdecim` applies scalar expansion as done for the word lengths.

Here is the `SpecifyPrecision` display.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyPrecision'

SectionWordLengths: [19 18 18 17]
SectionFracLengths: [14 13 13 12]

OutputWordLength: 16
OutputFracLength: 11
```

About the States of the Filter

In the states property you find the states for both the integrator and comb portions of the filter. states is a matrix of dimensions m + 1-by-n, with the states apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.

- States for the comb portion fill the remaining rows in the state matrix..

To review the states of a CIC filter, use int to assign the states to a variable in MATLAB. As an example, here are the states for a CIC decimator hm before and after filtering a data set.

```
x = fi(ones(1,10),true,16,0); % Fixed-point input data.
hm = mfilt.cicdecim(2,1,2,16,16,16);
sts=int(hm.states)
set(hm,'InputFracLength',0); % Integer input specified.
y=filter(hm,x);
sts=int(hm.states)
```

STS is an integer matrix that int returns from the contents of the filtstates.cic object in hm.

Design Considerations

When you design your CIC decimation filter, remember the following general points:

- The filter output spectrum has nulls at $\omega = k * 2\pi/rm$ radians, $k = 1,2,3....$

- Aliasing and imaging occur in the vicinity of the nulls.

- n, the number of sections in the filter, determines the passband attenuation. Increasing n improves the filter ability to reject aliasing and imaging, but it also increases the droop (or rolloff) in the filter passband. Using an appropriate FIR filter in series after the CIC decimation filter can help you compensate for the induced droop.

- The DC gain for the filter is a function of the decimation factor. Raising the decimation factor increases the DC gain.

Examples

This example applies a decimation factor r equal to 8 to a 160-point impulse signal. The signal output from the filter has 160/r, or 20, points or samples. Choosing 10 bits for the word length represents a fairly common setting for analog to digital converters. The plot shown after the code presents the stem plot of the decimated signal, with 20 samples remaining after decimation:

```
m = 2;  % Differential delays in the filter.
n = 4;  % Filter sections
r = 8;   % Decimation factor
x = int16(zeros(160,1)); x(1) = 1;    % Create a 160-point
                                      % impulse signal.
hm = mfilt.cicdecim(r,m,n); % Expects 16-bit input
                            % by default.
y = filter(hm,x);
stem(double(y));     % Plot output as a stem plot.
xlabel('Samples'); ylabel('Amplitude');
title('Decimated Signal');
```

Decimated Signal
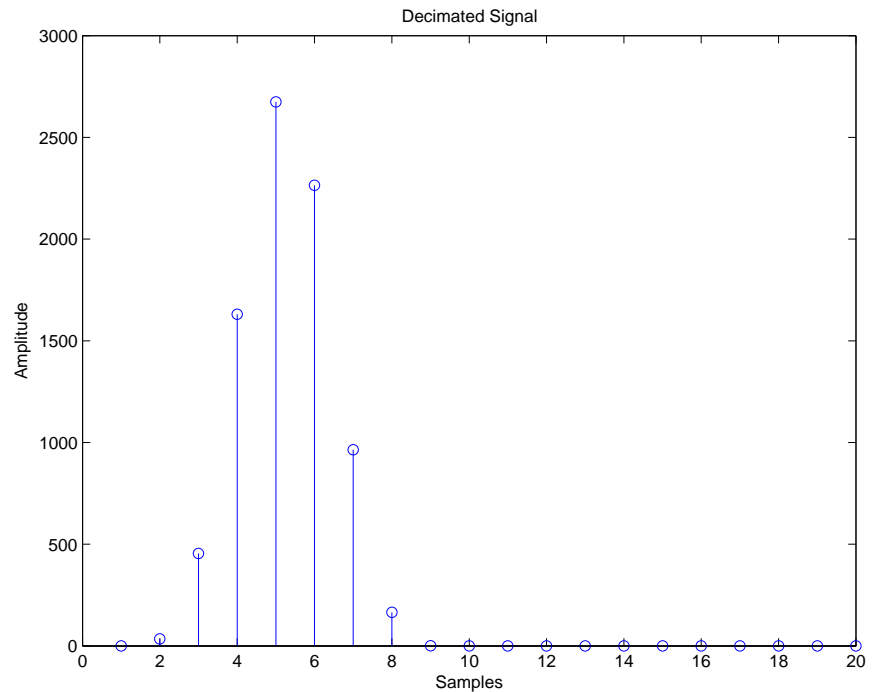
The next example demonstrates one way to compute the filter frequency response, using a 4-section decimation filter with the decimation factor set to 7:

```
hm = mfilt.cicdecim(7,1,4);
fvtool(hm)
```

FVTool provides ways for you to change the title and x labels to match the figure shown. Here's the frequency response plot for the filter. For details about the transfer function used to produce the frequency response, refer to [1] in the References section.
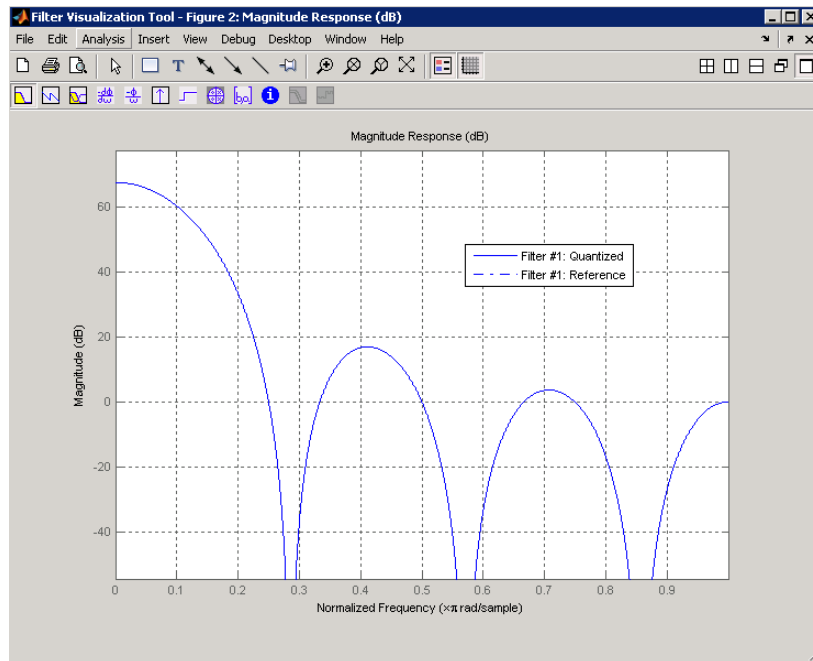
mfilt.cicdecim

This final example demonstrates the decimator for converting from 44.1 kHz audio to 22.05 kHz — decimation by two. To overlay the before and after signals, scale the output and plot the signals on a stem plot.

```
r = 2;                  % Decimation factor.
hm = mfilt.cicdecim(r); % Use default NumberOfSections &
                        % DifferentialDelay property values.
fs = 44.1e3;            % Original sampling frequency: 44.1kHz.
n = 0:10239;            % 10240 samples, 0.232 second long signal.
x  = sin(2*pi*1e3/fs*n);% Original signal, sinusoid at 1kHz.

y_fi = filter(hm,x); % 5120 samples, still 0.232 seconds.

% Scale the output to overlay the stem plots.
x = double(x);
y = double(y_fi);
```
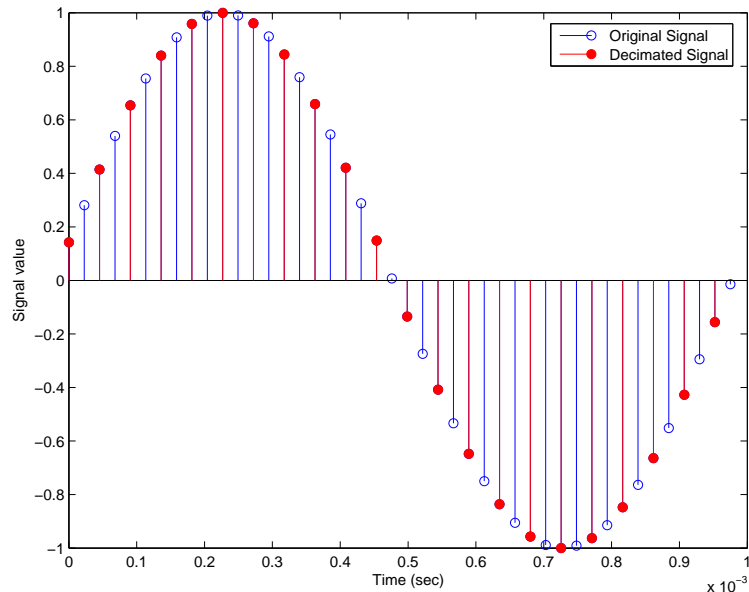
```
y = y/max(abs(y));
stem(n(1:44)/fs,x(2:45)); hold on;  % Plot original signal
                                    % sampled at 44.1kHz.
stem(n(1:22)/(fs/r),y(3:24),'r','filled'); % Plot decimated
                                    % signal (22.05kHz)
                                    % in red.
xlabel('Time (seconds)');ylabel('Signal Value');
```
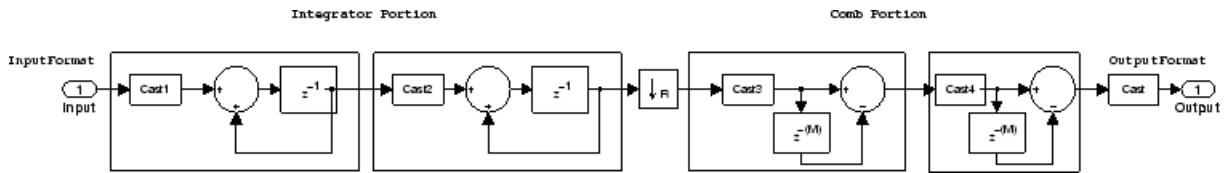

Algorithms   To show how the CIC decimation filter is constructed, the following figure presents a block diagram of the filter structure for a two-section CIC decimation filter (n = 2). *fs* is the high sampling rate, the input to the decimation process.

For details about the bits that are removed in the Comb section, refer to [1] in References.

mfilt.cicdecim

`mfilt.cicdecim` calculates the fraction length at each section of the decimator to avoid overflows at the output of the filter.

References [1] Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-29(2): pp. 155-162, 1981

[2] Meyer-Baese, Uwe, "Hogenauer CIC Filters," in *Digital Signal Processing with Field Programmable Gate Arrays*, Springer, 2001, pp. 155-172

[3] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice-Hall PTR, 2004 , pp. 343

See Also `mfilt` | `mfilt.cicinterp`

Purpose   Fixed-point CIC interpolator

Syntax   hm = mfilt.cicinterp(R,M,N,ILW,OWL,WLPS)
                     hm = mfilt.cicinterp
                     hm = mfilt.cicinterp(R,...)

Description   hm = mfilt.cicinterp(R,M,N,ILW,OWL,WLPS) constructs a cascaded
                    integrator-comb (CIC) interpolation filter object that uses fixed-point
                    arithmetic.

                    All of the input arguments are optional. To enter any optional value,
                    you must include all optional values to the left of your desired value.

                    When you omit one or more input options, the omitted option applies
                    the default values shown in the table below.

                    The following table describes the input arguments for creating hm.

Input Arguments	Description
R	Interpolation factor applied to the input signal. Sharpens the response curve to let you change the shape of the response. R must be an integer value greater than or equal to 1. The default value is 2.
M	Differential delay. Changes the shape, number, and location of nulls in the filter response. Increasing M increases the sharpness of the nulls and the response between nulls. In practice, differential delay values of 1 or 2 are the most common. M must be an integer value greater than or equal to 1. The default value is 1.
N	Number of sections. Deepens the nulls in the response curve. Note that this is the number of either comb or integrator sections, not the total section count. By default, the filter has two sections.

Input Arguments	Description
IWL	Word length of the input signal. Use any integer number of bits. The default value is 16 bits.
OWL	Word length of the output signal. It can be any positive integer number of bits. By default, OWL is 16 bits.
WLPS	Defines the number of bits per word in each filter section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter WLPS as a scalar or vector of length 2*N, where N is the number of sections. When WLPS is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the integrator. When you elect to specify wlps as an input argument, the FilterInternals property automatically switches from the default value of 'FullPrecision' to 'SpecifyWordLengths'.

hm = mfilt.cicinterp constructs the CIC interpolator using the default values for the optional input arguments.

hm = mfilt.cicinterp(R,...) constructs the CIC interpolator applying the values you provide for R and any other values you specify as input arguments.

Constraints and Conversions

In Hogenauer [1], the author describes the constraints on CIC interpolator filters. mfilt.cicinterp enforces a constraint—the word lengths of the filter sections must be non-decreasing. That is, the word length of each filter section must be the same size as, or greater than, the word length of the previous filter section.

The formula for W_j, the minimum register width, is derived in [1]. The formula for W_j is given by

$$W_j = ceil(B_{in} + \log_2 G_j)$$

where G_j, the maximum register growth up to the jth section, is given by

$$G_j = \begin{cases} 2^j, & j = 1, 2, \ldots, N \\ \dfrac{2^{2N-j}(RM)^{j-N}}{R}, & j = N+1, \ldots, 2N \end{cases}$$

When the differential delay, M, is 1, there is also a special condition for the register width of the last comb, W_N, that is given by

$$W_N = B_{in} + N - 1 \quad if \quad M = 1$$

The conversions denoted by the cast blocks in the integrator diagrams in "Algorithms" on page 7-1035 perform the changes between the word lengths of each section. When you specify word lengths that do not follow the constraints described in this section, mfilt.cicinterp returns an error.

The fraction lengths and scalings of the filter sections do not change. At each section the word length is either staying the same or increasing. The signal scaling can change at the output after the final filter section if you choose the output word length to be less than the word length of the final filter section.

Properties of the Object

The following table lists the properties for the filter with a description of each.

Name	Values	Default	Description
Arithmetic	`fixed`	`fixed`	Reports the kind of arithmetic the filter uses. CIC interpolators are always fixed-point filters.
InterpolationFactor	Any positive integer	2	Amount to increase the input sampling rate.
DifferentialDelay	Any positive integer	1	Sets the differential delay for the filter. Usually a value of one or two is appropriate.
FilterStructure	mfilt structure string	None	Reports the type of filter object, such as a interpolator or fractional integrator. You cannot set this property — it is always read only and results from your choice of `mfilt` objects.
FilterInternals	`FullPrecision`, `MinWordLengths`, `SpecifyWordLengths`, `SpecifyPrecision`	`FullPrecision`	Set the usage mode for the filter. Refer to "Usage Modes" on page 7-1031 below for details.
InputFracLength	Any positive integer	16	The number of bits applied as the fraction length to interpret the input data to the filter.

Name	Values	Default	Description
InputWordLength	Any positive integer	16	The number of bits applied to the word length to interpret the input data to the filter.
NumberOfSections	Any positive integer	2	Number of sections used in the interpolator. Generally called n. Reflects either the number of interpolator or comb sections, not the total number of sections in the filter.
OutputFracLength	Any positive integer	15	The number of bits applied to the fraction length to interpret the output data from the filter. Read-only.
OutputWordLength	Any positive integer	16	The number of bits applied to the word length to interpret the output data from the filter.
PersistentMemory	false or true	false	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing.

Name	Values	Default	Description
			States that the filter does not change are not affected. When `PersistentMemory` is `false`, you cannot access the filter states. Setting `PersistentMemory` to `true` reveals the `States` property so you can modify the filter states.
SectionWordLengths	Any integer or a vector of length 2^N, where N is a positive integer. This property only applies when the `FilterInternals` is `SpecifyWordLengths`.	16	Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter `SectionWordLengths` as a scalar or vector of length 2*n, where n is the number of sections. When `SectionWordLengths` is a scalar, the scalar value is applied to each filter section. When `SectionWordLengths` is a vector of values, the values apply to the sections in order. The default is 16 for each section in the interpolator. Available

Name	Values	Default	Description
			when `FilterInternals` is `'SpecifyWordLengths'`.
States	`filtstates.cic` object	m+1-by-n matrix of zeros, after you call function int.	Stored conditions for the filter, including values for the integrator and comb sections before and after filtering. m is the differential delay of the comb section and n is the number of sections in the filter. The integrator states are stored in the first matrix row. States for the comb section fill the remaining rows in the matrix. Available for modification when `PersistentMemory` is `true`. Refer to the `filtstates` object in Signal Processing Toolbox documentation for more general information about the `filtstates` object.

Usage Modes

There are usage modes which are set using the `FilterInternals` property:

- `FullPrecision` — In this mode, the word and fraction lengths of the filter sections and outputs are automatically selected for you. The output and last section word lengths are set to:

$$\text{wordlength} = \text{ceil}(\log_2((RM)^N / R)) + I,$$

where *R* is the interpolation factor, *M* is the differential delay, *N* is the number of filter sections, and *I* denotes the input word length.

- `MinWordLengths` — In this mode, you specify the word length of the filter output in the `OutputWordLength` property. The word lengths of the filter sections are automatically set in the same way as in the `FullPrecision` mode. The section fraction lengths are set to the input fraction length. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.

- `SpecifyWordLengths` — In this mode, you specify the word lengths of the filter sections and output in the `SectionWordLengths` and `OutputWordLength` properties. The fraction lengths of the filter sections are set such that the spread between word length and fraction length is the same as in full-precision mode. The output fraction length is set to the input fraction length minus the difference between the last section and output word lengths.

- `SpecifyPrecision` — In this mode, you specify the word and fraction lengths of the filter sections and output in the `SectionWordLengths`, `SectionFracLengths`,`OutputWordLength`, and `OutputFracLength` properties.

About the States of the Filter

In the `states` property you find the states for both the integrator and comb portions of the filter. `states` is a matrix of dimensions `m+1-by-n`, with the states apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.

- States for the comb portion fill the remaining rows in the state matrix.

mTo review the states of a CIC filter, use the `int` method to assign the states. As an example, here are the states for a CIC interpolator `hm` before and after filtering data:

```
x = fi(cos(pi/4*[0:99]),true,16,0); % Fixed-point input data
```

```
hm = mfilt.cicinterp(2,1,2,16,16,16);
% get initial states-all zero
sts=int(hm.states)
set(hm,'InputFracLength',0); % Integer input specified
y=filter(hm,x);
sts=int(hm.states)
%sts =
%
%    -1    -1
%    -1    -1
```

Design Considerations

When you design your CIC interpolation filter, remember the following general points:

- The filter output spectrum has nulls at $\omega = k * 2\pi/rm$ radians, $k = 1,2,3....$

- Aliasing and imaging occur in the vicinity of the nulls.

- n, the number of sections in the filter, determines the passband attenuation. Increasing n improves the filter ability to reject aliasing and imaging, but it also increases the droop or rolloff in the filter passband. Using an appropriate FIR filter in series after the CIC interpolation filter can help you compensate for the induced droop.

- The DC gain for the filter is a function of the interpolation factor. Raising the interpolation factor increases the DC gain.

Examples   Demonstrate interpolation by a factor of two, in this case from 22.05 kHz to 44.1 kHz. Note the scaling required to see the results in the stem plot and to use the full range of the int16 data type.

```
R = 2;                    % Interpolation factor.
hm = mfilt.cicinterp(R);  % Use default NumberOfSections and
                          % DifferentialDelay property values.
fs = 22.05e3;             % Original sample frequency:22.05 kHz.
n = 0:5119;               % 5120 samples, .232 second long signal.
x = sin(2*pi*1e3/fs*n);   % Original signal, sinusoid at 1 kHz.
```

```
y_fi = filter(hm,x); % 5120 samples, still 0.232 seconds.

% Scale the output to overlay stem plots correctly.
x = double(x);
y = double(y_fi);
y = y/max(abs(y));
stem(n(1:22)/fs,x(1:22),'filled'); % Plot original signal sampled
                                   % at 22.05 kHz.
hold on;
stem(n(1:44)/(fs*R),y(4:47),'r');  % Plot interpolated signal
                                   % (44.1 kHz) in red.
xlabel('Time (sec)');ylabel('Signal Value');
```

As you expect, the plot shows that the interpolated signal matches the input sine shape, with additional samples between each original sample.
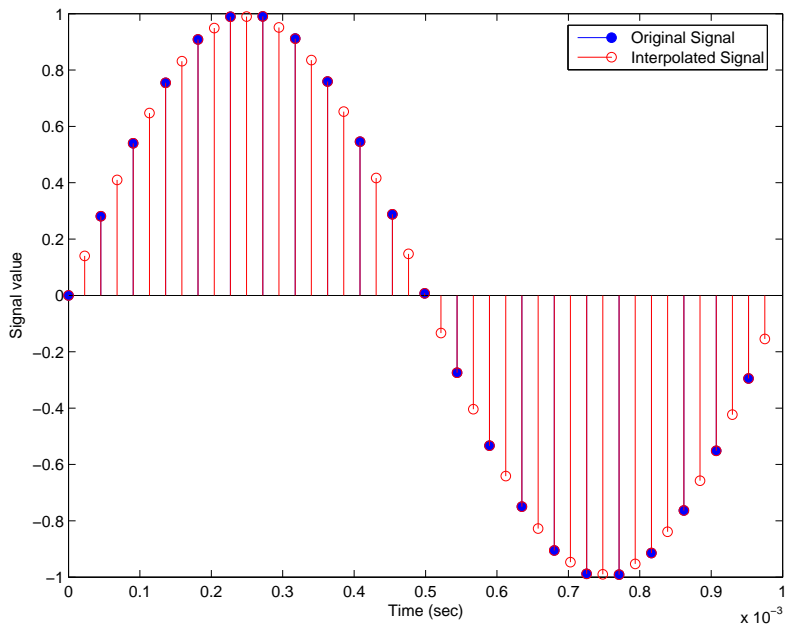
Use the filter visualization tool (FVTool) to plot the response of the interpolator object. For example, to plot the response of an interpolator with an interpolation factor of 7, 4 sections, and 1 differential delay, do something like the following:

```
hm = mfilt.cicinterp(7,1,4)
fvtool(hm)
```

Algorithms To show how the CIC interpolation filter is constructed, the following figure presents a block diagram of the filter structure for a two-section CIC interpolation filter (n = 2). *fs* is the high sampling rate, the output from the interpolation process.

For details about the bits that are removed in the integrator section, refer to [1] in References.
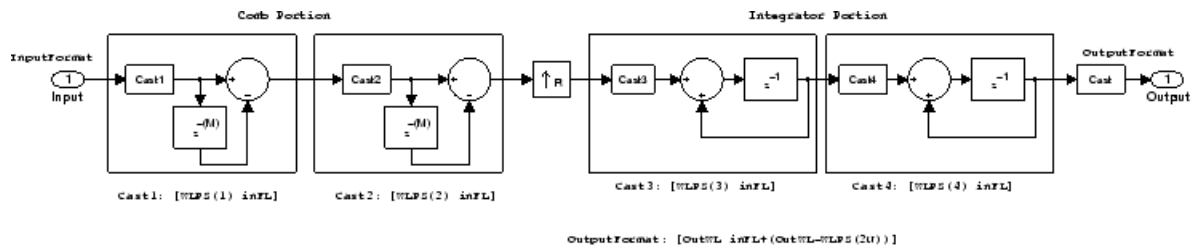
When you select MinWordLengths, the filter section word lengths are automatically set to the minimum number of bits possible in a valid CIC interpolator. mfilt.cicinterp computes the wordlength for each section so the roundoff noise introduced by all sections is less than the roundoff noise introduced by the quantization at the output.

References [1] Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-29(2): pp. 155-162, 1981

[2] Meyer-Baese, Uwe, "Hogenauer CIC Filters," in Digital Signal Processing with Field Programmable Gate Arrays, Springer, 2001, pp. 155-172

[3] Harris, Fredric J., *Multirate Signal Processing for Communication Systems*, Prentice-Hall PTR, 2004 , pp. 343

Purpose   Sample rate converter with arbitrary conversion factor

Syntax   hm = mfilt.farrowsrc(L,M,C)
                   hm = mfilt.farrowsrc
                   hm = mfilt.farrowsrc(l,...)

Description   hm = mfilt.farrowsrc(L,M,C) returns a filter object that is a natural
                   extension of dfilt.farrowfd with a time-varying fractional delay. It
                   provides a economical implementation of a sample rate converter
                   with an arbitrary conversion factor. This filter works well in the
                   interpolation case, but may exhibit poor anti-aliasing properties in the
                   decimation case.

Note You can use the realizemdl method to create a Simulink block
of a filter created using mfilt.farrowsrc.

Input Arguments

The following table describes the input arguments for creating hm.

Input Argument	Description
l	Interpolation factor for the filter. l specifies the amount to increase the input sampling rate. The default value of l is 3.
m	Decimation factor for the filter. m specifies the amount to decrease the input sampling rate. The default value for m is 2.
c	Coefficients for the filter. When no input arguments are specified, the default coefficients are [-1 1; 1, 0]

hm = mfilt.farrowsrc constructs the filter using the default values
for l, m, and c.

hm = mfilt.farrowsrc(l,...) constructs the filter using the input arguments you provide and defaults for the argument you omit.

mfilt.farrowsrc Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.farrowsrc objects. The next table describes each property for an mfilt.farrowsrc filter object.

| Name	Values	Description
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object.
Arithmetic	String	Reports the arithmtetic precision used by the filter.
Coefficients	Vector	Vector containing the coefficients of the FIR lowpass filter
InterpolationFactor	Integer	Interpolation factor for the filter. It specifies the amount to increase the input sampling rate.

Name	Values	Description
DecimationFactor	Integer	Decimation factor for the filter. It specifies the amount to increase the input sampling rate.
PersistentMemory	false or true	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.

Examples   Interpolation by a factor of 8. This object removes the spectral replicas in the signal after interpolation.

```
[L,M] = rat(48/44.1);
Hm = mfilt.farrowsrc(L,M);          % We use the default filter
Fs = 44.1e3;                        % Original sampling frequency
n = 0:9407;                         % 9408 samples, 0.213 seconds long
x = sin(2*pi*1e3/Fs*n);            % Original signal, sinusoid at 1kHz
y = filter(Hm,x);                   % 10241 samples, still 0.213 seconds
stem(n(1:45)/Fs,x(1:45))            % Plot original sampled at 44.1kHz
hold on
% Plot fractionally interpolated signal (48kHz) in red
stem((n(2:50)-1)/(Fs*L/M),y(2:50),'r','filled')
xlabel('Time (sec)');ylabel('Signal value')
legend('44.1 kHz sample rate','48kHz sample rate')
```

The results of the example are shown in the following figure:

mfilt.farrowsrc
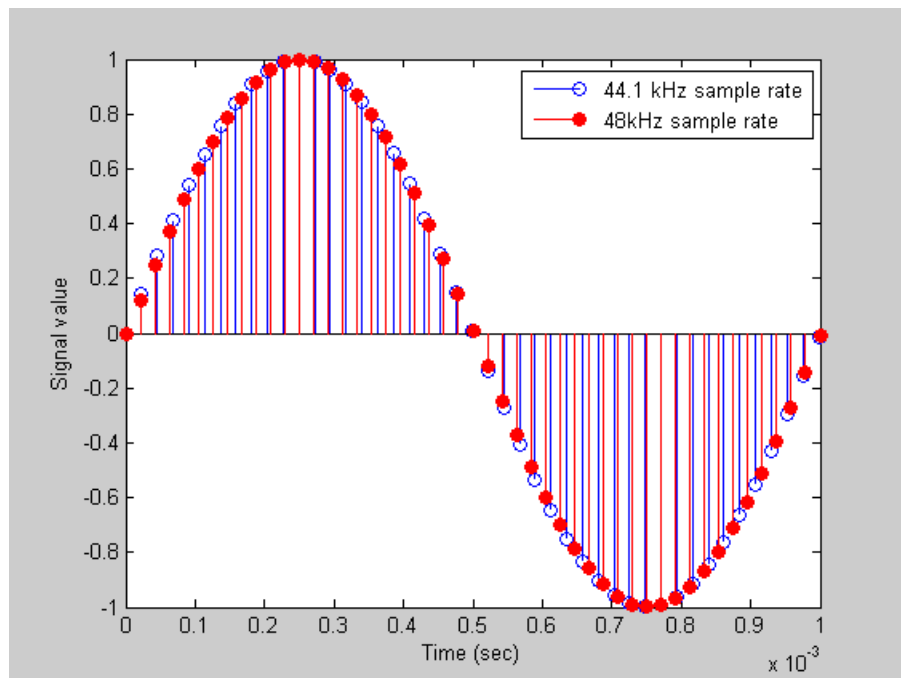
Purpose   Overlap-add FIR polyphase interpolator

Syntax   hm = mfilt.fftfirinterp(l,num,bl)
                 hm = mfilt.fftfirinterp
                 hm = mfilt.fftfirinterp(l,...)

Description hm = mfilt.fftfirinterp(l,num,bl) returns a discrete-time FIR
                 filter object that uses the overlap-add method for filtering input data.

The input arguments are optional. To enter any optional value, you
must include all optional values to the left of your desired value.

When you omit one or more input options, the omitted option applies
the default values shown in the table below.

The number of FFT points is given by [bl+ceil(length(num)/l)-1]. It
is to your advantage to choose bl such that the number of FFT points is
a power of two—using powers of two can improve the efficiency of the
FFT and the associated interpolation process.

Input Arguments

The following table describes the input arguments for creating hm.

Input Argument	Description
l	Interpolation factor for the filter. l specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for l it defaults to 2.
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, fftfirinterp uses a lowpass Nyquist filter with gain equal to l and cutoff frequency equal to π/l by default.
bl	Length of each block of input data used in the filtering. bl must be an integer. When you omit input bl, it defaults to 100

hm = `mfilt.fftfirinterp` constructs the filter using the default values for `l`, `num`, and `bl`.

hm = `mfilt.fftfirinterp(l,...)` constructs the filter using the input arguments you provide and defaults for the argument you omit.

mfilt.fftfirinterp Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.fftfirinterp` objects. The next table describes each property for an `mfilt.fftfirinterp` filter object.

Name	Values	Description
FilterStructure		Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
Numerator		Vector containing the coefficients of the FIR lowpass filter used for interpolation.
InterpolationFactor		Interpolation factor for the filter. It specifies the amount to increase the input sampling rate. It must be an integer.
BlockLength		Length of each block of input data used in the filtering.

Name	Values	Description
PersistentMemory	false or true	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.
States		Stored conditions for the filter, including values for the interpolator states.

Examples   Interpolation by a factor of 8. This object removes the spectral replicas in the signal after interpolation.

```
l = 8;                         % Interpolation factor
hm = mfilt.fftfirinterp(l);    % We use the default filter
n = 8192;                      % Number of points
hm.blocklength = n;            % Set block length to number of points
fs = 44.1e3;                   % Original sample freq: 44.1 kHz.
n = 0:n-1;                     % 0.1858 secs of data
x = sin(2*pi*n*22e3/fs);       % Original signal, sinusoid at 22 kHz
y = filter(hm,x);              % Interpolated sinusoid
xu = l*upsample(x,8);          % Upsample to compare--the spectrum
                               % does not change
[px,f]=periodogram(xu,[],65536,l*fs);% Power spectrum of original
                                     % signal
[py,f]=periodogram(y,[],65536,l*fs); % Power spectrum of
                                     % interpolated signal
```
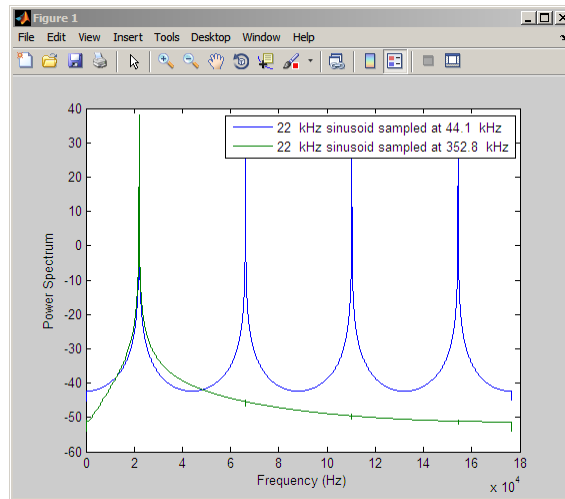
```
plot(f,10*log10(([fs*px,l*fs*py])))
legend('22  kHz sinusoid sampled at 44.1  kHz',...
'22  kHz sinusoid sampled at 352.8  kHz')
xlabel('Frequency (Hz)'); ylabel('Power Spectrum');
```

To see the results of the example, look at this figure.

See Also   mfilt.firinterp | mfilt.holdinterp | mfilt.linearinterp | mfilt.firsrc | mfilt.cicinterp

Purpose   Direct-form FIR polyphase decimator

Syntax   `hm = mfilt.firdecim(m)`
                   `hm = mfilt.firdecim(m,num)`

Description `hm = mfilt.firdecim(m)` returns a direct-form FIR polyphase decimator object `hm` with a decimation factor of `m`. A lowpass Nyquist filter of gain 1 and cutoff frequency of π/m is designed by default. This filter allows some aliasing in the transition band but it very efficient because the first polyphase component is a pure delay.

`hm = mfilt.firdecim(m,num)` uses the coefficients specified by `num` for the decimation filter. This lets you specify more completely the FIR filter to use for the decimator.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  `set(hm,'arithmetic','single');`

- To change to fixed-point filtering, enter

  `set(hm,'arithmetic','fixed');`

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to 2.
num	Vector containing the coefficients of the FIR lowpass filter used for decimation. When num is not provided as an input, mfilt.firdecim constructs a lowpass Nyquist filter with gain of 1 and cutoff frequency equal to π/m by default. The default length for the Nyquist filter is 24*m. Therefore, each polyphase filter component has length 24.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firdecim objects. The next table describes each property for an mfilt.firdecim filter object.

Name	Values	Description
Arithmetic	Double, single, fixed	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operation mode for your filter.
DecimationFactor	Integer	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of

Name	Values	Description
		the input signal. It must be an integer.
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object. Describes the signal flow for the filter object.
InputOffset	Integers	Contains a value derived from the number of input samples and the decimation factor — InputOffset = mod(length(nx),m) where nx is the number of input samples that have been processed so far and m is the decimation factor.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.
PersistentMemory	false, true	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory set to false returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to

Name	Values	Description
		`true` allows you to modify the `States`, `InputOffset`, and `PolyphaseAccum` properties.
`PolyphaseAccum`	`0` in `double`, `single`, or `fixed` for the different filter arithmetic settings.	Differentiates between the adders in the filter that work in full precision at all times (`PolyphaseAccum`) and the adders in the filter that the user controls and that may introduce quantization effects when `FilterInternals` is set to `SpecifyPrecision`.
`States`	`Double`, `single`, or `fi` matching the filter arithmetic setting.	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Double is the default setting for floating-point filters in double arithmetic.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the filter. You see one or more of these properties when you set Arithmetic to fixed. Some of the properties have different default values when they refer fixed point filters. One example is the property `PolyphaseAccum` which stores data as doubles when you use your filter in double-precision mode, but stores a `fi` object in fixed-point mode.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use info(hm) where hm is a filter.

For further information about the properties of this filter or any mfilt object, refer to "Multirate Filter Properties" on page 8-115.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters.
AccumWordLength	Any integer number of bits [39]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to fixed allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

Name	Values	Description
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits[16]	Specifies the word length applied to interpret input data.
OutputFracLength	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilterInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

| Name	Values	Description
		The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object	This property contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation. For information about the ordering of the states, refer to the filter structure section.

Filter Structure

To provide decimation, mfilt.firdecim uses the following structure. At the input you see a commutator that operates counterclockwise, moving from position 0 to position 2, position 1, and back to position 0 as input samples enter the filter.

The following figure details the signal flow for the direct form FIR filter implemented by mfilt.firdecim.
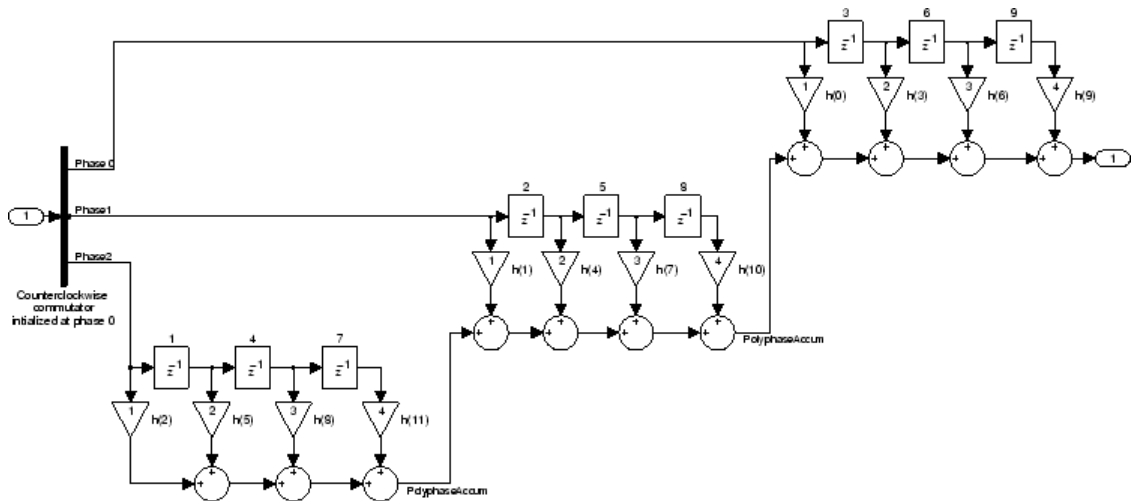
Notice the order of the states in the filter flow diagram. States 1 through 9 appear in the diagram above each delay element. State 1 applies to the first delay element in phase 2. State 2 applies to the first delay element in phase 1. State 3 applies to the first delay element in phase 0. State 4 applies to the second delay in phase 2, and so on. When you provide the states for the filter as a vector to the States property, the above description explains how the filter assigns the states you specify.

In property value form, the states for a filter hm are

```
hm.states=[1:9];
```

Examples   Convert an input signal from 44.1 kHz to 22.05 kHz using decimation by a factor of 2. In the figure that appears after the example code, you see the results of the decimation.

```
m = 2;                    % Decimation factor.
hm = mfilt.firdecim(m);   % Use the default filter.
fs = 44.1e3;              % Original sample freq: 44.1kHz.
n = 0:10239;              % 10240 samples, 0.232 second long
                          % signal.
x  = sin(2*pi*1e3/fs*n);  % Original signal--sinusoid at 1kHz.
```

```
y = filter(hm,x);              % 5120 samples, 0.232 seconds.
stem(n(1:44)/fs,x(1:44))       % Plot original sampled at 44.1 kHz.
hold on                        % Plot decimated signal (22.05 kHz)
                               % in red.
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```
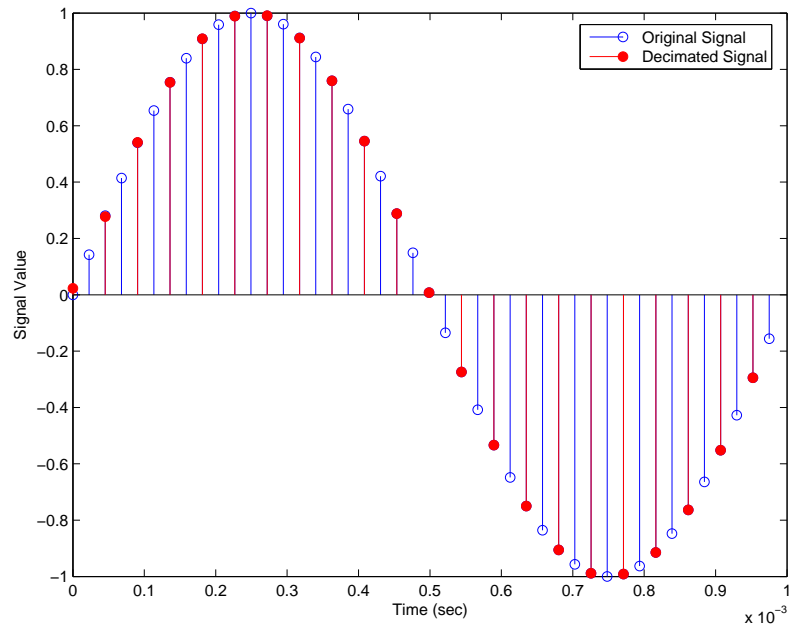

See Also   mfilt.firtdecim | mfilt.firsrc | mfilt.cicdecim

Purpose   Direct-form FIR polyphase fractional decimator

Syntax   hm = mfilt.firfracdecim(l,m,num)

Description mfilt.firfracdecim will be removed in a future release. Use
mfilt.firsrc instead.

hm = mfilt.firfracdecim(l,m,num) returns a direct-form FIR
polyphase fractional decimator. Input argument l is the interpolation
factor. l must be an integer. When you omit l in the calling syntax, it
defaults to 2. m is the decimation factor. It must be an integer. If not
specified, it defaults to l+1.

num is a vector containing the coefficients of the FIR lowpass filter used
for decimation. If you omit num, a lowpass Nyquist filter of gain l and
cutoff frequency of π/max(l,m) is used by default.

By specifying both a decimation factor and an interpolation factor, you
can decimate your input signal by noninteger amounts. The fractional
decimator first interpolates the input, then decimates to result in an
output signal whose sample rate is l/m of the input rate. By default, the
resulting decimation factor is 2/3 when you do not provide l and m in
the calling syntax. Specify l smaller than m for proper decimation.

Input Arguments

The following table describes the input arguments for creating hm.

Input Argument	Description
l	Interpolation factor for the filter. It must be an integer. When you do not specify a value for l it defaults to 2.
num	Vector containing the coefficients of the FIR lowpass filter used for decimation. When num is not provided as an input, firfracdecim uses a lowpass Nyquist

mfilt.firfracdecim (Obsolete)

Input Argument	Description
	filter with gain equal to l and cutoff frequency equal to π/max(l,m) by default.
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to l + 1.

mfilt.firfracdecim Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firfracdecim objects. The next table describes each property for an mfilt.firfracdecim filter object.

Name	Values	Description
FilterStructure	String	Reports the type of filter object, such as a decimator or fractional decimator. You cannot set this property — it is always read only and results from your choice of mfilt object.
InputOffset	Integers	Contains the number of input data samples processed without generating an output sample. The default value is 0.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for interpolation.

| Name	Values	Description
RateChangeFactors	[l,m]	Reports the decimation (m) and interpolation (l) factors for the filter object. Combining these factors results in the final rate change for the signal.
PersistentMemory	false or true	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.
States	Matrix	Stored conditions for the delays between each interpolator phase, the filter states, and the states at the output of each phase in the filter.
		The number of states is (lh-1)*m+(l-1)*(lo+mo) where lh is the length of each subfilter, and l and m are the interpolation and decimation factors. lo and mo, the input and output delays between each interpolation phase, are integers from Euclid's theorem such that lo*l-mo*m = -1 (refer to the reference for more details). Use euclidfactors to get lo and mo for an mfilt.firfracdecim object

mfilt.firfracdecim (Obsolete)

Examples

To demonstrate `firfracdecim`, perform a fractional decimation by a factor of 2/3. This is one way to downsample a 48 kHz signal to 32 kHz, commonly done in audio processing.

```
l  = 2; m = 3;                    % Interpolation/decimation factors.
hm = mfilt.firfracdecim(l,m);     % We use the default
fs = 48e3;                        % Original sample freq: 48 kHz.
n  = 0:10239;                     % 10240 samples, 0.213 second long
                                  % signal
x  = sin(2*pi*1e3/fs*n);          % Original signal, sinusoid at 1 kHz
y  = filter(hm,x);                % 9408 samples, still 0.213 seconds
stem(n(1:49)/fs,x(1:49)); hold on; % Plot original signal sampled
                                      % at 48 kHz
stem(n(1:32)/(fs*l/m),y(13:44),'r','filled') % Plot decimated
                                                % signal at 32 kHz
xlabel('Time (sec)');
```

As shown, the plot clearly demonstrates the reduced sampling frequency of 32 kHz.
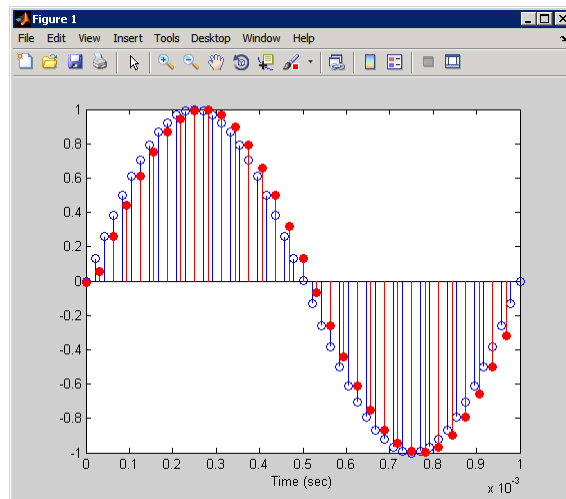
References   Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley & Sons, Ltd., 1994

See Also   `mfilt.firsrc` | `mfilt.firinterp` | `mfilt.firdecim`

mfilt.firfracinterp (Obsolete)

Purpose   Direct-form FIR polyphase fractional interpolator

Syntax   `hm = mfilt.firfracinterp(l,m,num)`

Description   `mfilt.firfracinterp` will be removed in a future release. Use `mfilt.firsrc` instead.

`hm = mfilt.firfracinterp(l,m,num)` returns a direct-form FIR polyphase fractional interpolator mfilt object. `l` is the interpolation factor. It must be an integer. If not specified, `l` defaults to 3.

`m` is the decimation factor. Like `l`, it must be an integer. If you do not specify `m` in the calling syntax, it defaults to 1. If you also do not specify a value for `l`, `m` defaults to 2.

`num` is a vector containing the coefficients of the FIR lowpass filter used for interpolation. If omitted, a lowpass Nyquist filter of gain `l` and cutoff frequency of π/`max(l,m)` is used by default.

By specifying both a decimation factor and an interpolation factor, you can interpolate your input signal by noninteger amounts. The fractional interpolator first interpolates the input, then decimates to result in an output signal whose sample rate is `l`/`m` of the input rate. For proper interpolation, you specify `l` to be greater than `m`. By default, the resulting interpolation factor is 3/2 when you do not provide `l` and `m` in the calling syntax.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
l	Interpolation factor for the filter. l specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for l it defaults to 3.
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, firfracinterp uses a lowpass Nyquist filter with gain equal to l and cutoff frequency equal to $\pi/\max(l,m)$ by default.
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to 1. When you do not specify l as well, m defaults to 2.

mfilt.firfracinterp Object Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firfracinterp objects. The next table describes each property for an mfilt.firfracinterp filter object.

Name	Values	Description
FilterStructure		Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object.
Numerator		Vector containing the coefficients of the FIR lowpass filter used for interpolation.

mfilt.firfracinterp (Obsolete)

Name	Values	Description
RateChangeFactors	[l,m]	Reports the decimation (m) and interpolation (l) factors for the filter object. Combining these factors results in the final rate change for the signal.
PersistentMemory	false or true	Determines whether the filter states are restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory returns to the default values any state that the filter changes during processing. States that the filter does not change are not affected.
States	Matrix	Stored conditions for the filter, including values for the interpolator and comb states.

Examples To convert a signal from 32 kHz to 48 kHz requires fractional interpolation. This example uses the mfilt.firfracinterp object to upsample an input signal. Setting l = 3 and m = 2 returns the same mfilt object as the default mfilt.firfracinterp object.

```
l  = 3; m = 2;            % Interpolation/decimation factors.
hm = mfilt.firfracinterp(l,m);    % We use the default filter
fs = 32e3;               % Original sample freq: 32 kHz.
n = 0:6799;              % 6800 samples, 0.212 second long signal
x  = sin(2*pi*1e3/fs*n);   % Original signal, sinusoid at 1 kHz
```

```
y = filter(hm,x);              % 10200 samples, still 0.212 seconds
stem(n(1:32)/fs,x(1:32),'filled') % Plot original sampled at
                                    % 32 kHz
hold on;
% Plot fractionally interpolated signal (48 kHz) in red
stem(n(1:48)/(fs*l/m),y(20:67),'r')
xlabel('Time (sec)');ylabel('Signal Value')
```

The ability to interpolate by fractional amounts lets you raise the sampling rate from 32 to 48 kHz, something you cannot do with integral interpolators. Both signals appear in the following figure.
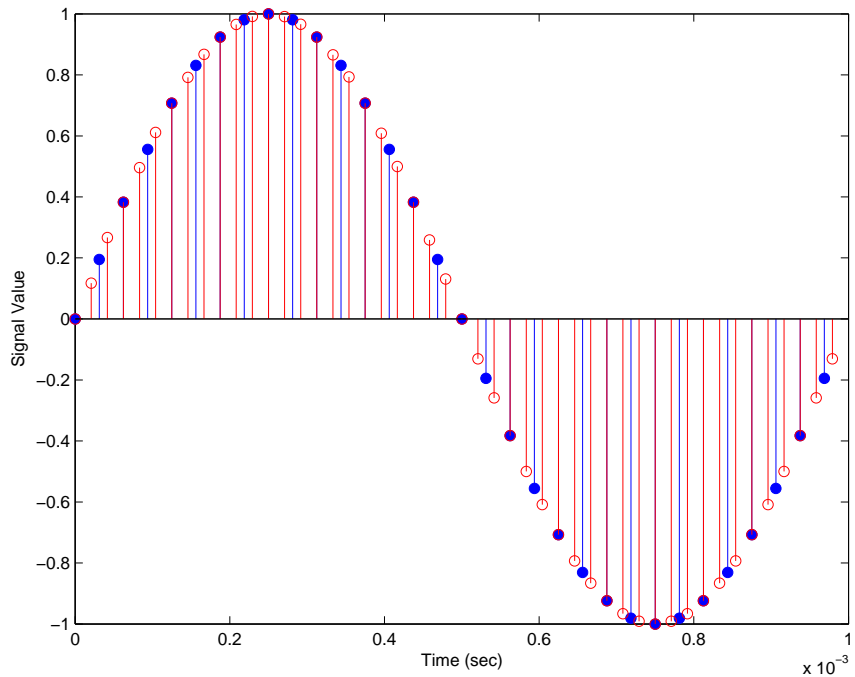
See Also   mfilt.firsrc | mfilt.firinterp | mfilt.firdecim

Purpose	FIR filter-based interpolator

Syntax

```
Hm = mfilt.firinterp(L)
Hm = mfilt.firinterp(L,num)
```

Description

`Hm = mfilt.firinterp(L)` returns a FIR polyphase interpolator object `Hm` with an interpolation factor of `L` and gain equal to `L`. `L` defaults to 2 if unspecified.

`Hm = mfilt.firinterp(L,num)` uses the values in the vector `num` as the coefficients of the interpolation filter.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `Hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
l	Interpolation factor for the filter. l specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for l it defaults to 2.
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, firinterp uses a lowpass Nyquist filter with gain equal to l and cutoff frequency equal to π/l by default. The default length for the Nyquist filter is 24*l. Therefore, each polyphase filter component has length 24.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.firinterp` objects. The next table describes each property for an `mfilt.firinterp` filter object.

Name	Values	Description
Arithmetic	Double, single, fixed	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operation mode for your filter.
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object. Describes the signal flow for the filter object.
InterpolationFactor	Integer	Interpolation factor for the filter. 1 specifies the amount to increase the sampling rate of the input signal. It must be an integer.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.

Name	Values	Description
PersistentMemory	[false], true	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory set to false returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to true allows you to modify the States property.
States	Double, single, matching the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the mfilt.firinterp filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where hm is a filter.

For further information about the properties of this filter or any mfilt object, refer to "Multirate Filter Properties" on page 8-115.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits. [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.
AccumWordLength	Any integer number of bits[39]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to fixed allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

Name	Values	Description
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilterInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

| Name	Values	Description
		The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object to match the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation.

Filter Structure

To provide interpolation, mfilt.firinterp uses the following structure.

The following figure details the signal flow for the direct form FIR filter implemented by mfilt.firinterp. In the figure, the delay line updates happen at the lower input rate. The remainder of the filter — the sums and coefficients — operate at the higher output rate.
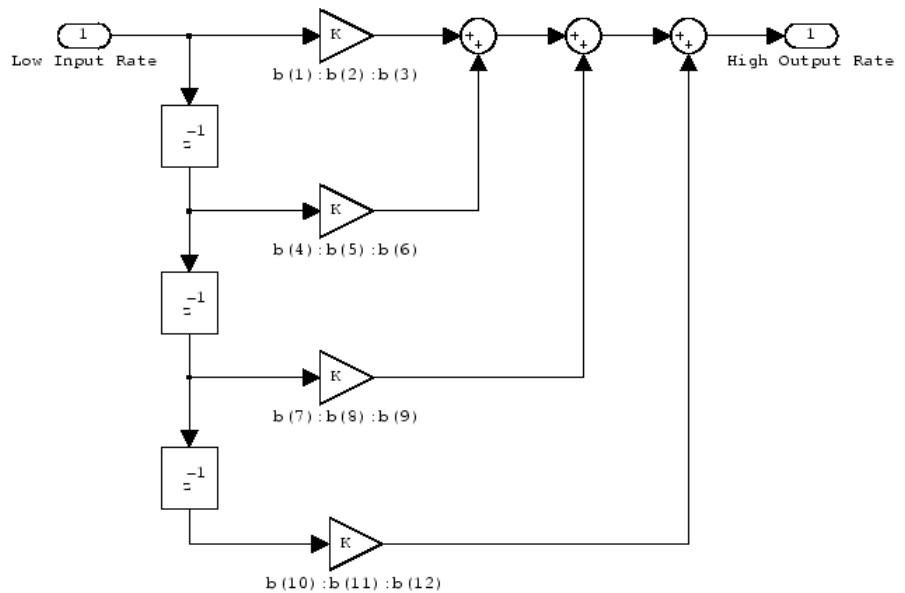
Examples   This example uses `mfilt.firinterp` to double the sample rate of a 22.05 kHz input signal. The output signal ends up at 44.1 kHz. Although l is set explicitly to 2, this represents the default interpolation value for `mfilt.firinterp` objects.

```
L = 2;                    % Interpolation factor.
Hm = mfilt.firinterp(L);  % Use the default filter.
fs = 22.05e3;             % Original sample freq: 22.05 kHz.
n = 0:5119;               % 5120 samples, 0.232s long signal.
x  = sin(2*pi*1e3/fs*n);  % Original signal, sinusoid at 1 kHz.
y = filter(Hm,x);         % 10240 samples, still 0.232s.
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
                                  % 22.05 kHz.

hold on;

% Plot interpolated signal (44.1 kHz) in red
stem(n(1:44)/(fs*L),y(25:68),'r')
xlabel('Time (sec)');ylabel('Signal Value')
```

```
legend('Original Signal','Interpolated Signal');
```

With interpolation by 2, the resulting signal perfectly matches the
original, but with twice as many samples — one between each original
sample, as shown in the following figure.
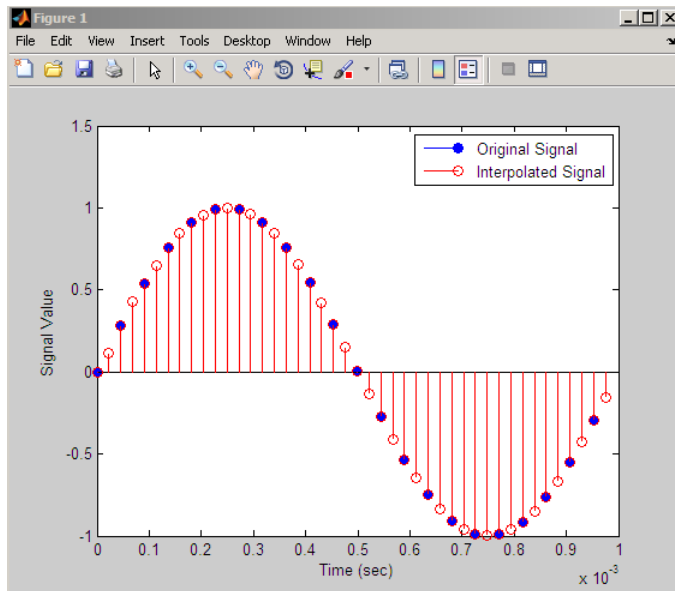
See Also   mfilt.holdinterp | mfilt.linearinterp | mfilt.fftfirinterp |
mfilt.firsrc | mfilt.cicinterp

Purpose   Direct-form FIR polyphase sample rate converter

Syntax   `hm = mfilt.firsrc(l,m,num)`

Description   `hm = mfilt.firsrc(l,m,num)` returns a direct-form FIR polyphase sample rate converter. `l` specifies the interpolation factor. It must be an integer and when omitted in the calling syntax, it defaults to 2.

`m` is the decimation factor. It must be an integer. If not specified, `m` defaults to 1. If `l` is also not specified, `m` defaults to 3 and the overall rate change factor is 2/3.

You specify the coefficients of the FIR lowpass filter used for sample rate conversion in `num`. If omitted, a lowpass Nyquist filter with gain `l` and cutoff frequency of $\pi/\texttt{max}(\texttt{l,m})$ is the default.

Combining an interpolation factor and a decimation factor lets you use `mfilt.firsrc` to perform fractional interpolation or decimation on an input signal. Using an `mfilt.firsrc` object applies a rate change factor defined by `l/m` to the input signal. For proper rate changing to occur, `l` and `m` must be relatively prime — meaning the ratio `l/m` cannot be reduced to a ratio of smaller integers.

When you are doing sample-rate conversion with large values of `l` or `m`, such as `l` or `m` greater than 20, using the `mfilt.firsrc` structure is the most effective approach.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

> **Note** You can use the `realizemdl` method to create a Simulink block of a filter created using `mfilt.firsrc`.

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
l	Interpolation factor for the filter. l specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for l, it defaults to 2.
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, mfilt.firsrc uses a lowpass Nyquist filter with gain equal to l and cutoff frequency equal to π/max(l,m) by default. The default length for the Nyquist filter is 24*m. Therefore, each polyphase filter component has length 24.
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m, it defaults to 1. When l is unspecified as well, m defaults to 3.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also

input arguments for creating `mfilt.firsrc` objects. The next table describes each property for an `mfilt.firsrc` filter object.

Name	Values	Description
Arithmetic	[Double], single, fixed	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operation mode for your filter.
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object. Describes the signal flow for the filter object.
InputOffset	Integers	Contains a value derived from the number of input samples and the decimation factor — InputOffset = mod(length(nx),m) where nx is the number of input samples and m is the decimation factor.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.
PersistentMemory	false, true	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. PersistentMemory set to false returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to true allows you to modify the States, InputOffset, and PolyphaseAccum properties.

Name	Values	Description
RateChangeFactors	Positive integers. [2 3]	Specifies the interpolation and decimation factors [l m] (the rate change factors) for changing the input sample rate by nonintegral amounts.
States	Double, single, matching the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the mfilt.firsrc filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

info(hm)

where hm is a filter.

For further information about the properties of this filter or any mfilt object, refer to "Multirate Filter Properties" on page 8-115.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits. [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters.
AccumWordLength	Any integer number of bits [39]	Sets the word length used to store data in the accumulator.

Name	Values	Description
Arithmetic	fixed for fixed-point filters	Setting this to fixed allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.

Name	Values	Description
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilterInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
RateChangeFactors	Positive integers [2 3]	Specifies the interpolation and decimation factors [1 m] (the rate change factors) for changing the input sample rate by nonintegral amounts.

Name	Values	Description
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation. For information about the ordering of the states, refer to the filter structure section.

Examples   This is an example of a common audio rate change process — changing the sample rate of a high end audio (48 kHz) signal to the compact disc sample rate (44.1 kHz). This conversion requires a rate change factor of 0.91875, or l = 147 and m = 160.

```
l  = 147; m = 160;          % Interpolation/decimation factors.
hm = mfilt.firsrc(l,m);     % Use the default FIR filter.
fs = 48e3;                  % Original sample freq: 48 kHz.
n = 0:10239;                % 10240 samples, 0.213 seconds long.
x  = sin(2*pi*1e3/fs*n);    % Original signal, sinusoid at 1 kHz.
y = filter(hm,x);           % 9408 samples, still 0.213 seconds.
stem(n(1:49)/fs,x(1:49))    % Plot original sampled at 48 kHz.
hold on

% Plot fractionally decimated signal (44.1 kHz) in red
stem(n(1:45)/(fs*l/m),y(13:57),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value')
```

Fractional decimation provides you the flexibility to pick and choose the sample rates you want by carefully selecting l and m, the interpolation and decimation factors, that result in the final fractional decimation.

The following figure shows the signal after applying the rate change filter `hm` to the original signal.
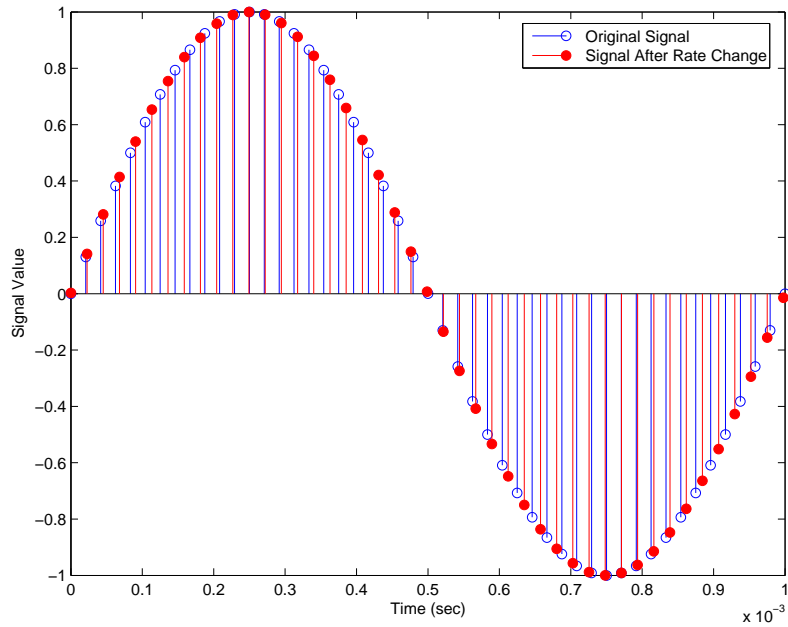
See Also   `mfilt.firsrc` | `mfilt.firinterp` | `mfilt.firdecim`

mfilt.firtdecim

Purpose   Direct-form transposed FIR filter

Syntax
```
hm = mfilt.firtdecim(m)
hm = mfilt.firtdecim(m,num)
```

Description   `hm = mfilt.firtdecim(m)` returns a polyphase decimator `mfilt` object `hm` based on a direct-form transposed FIR structure with a decimation factor of `m`. A lowpass Nyquist filter of gain 1 and cutoff frequency of π/m is the default.

`hm = mfilt.firtdecim(m,num)` uses the coefficients specified by `num` for the decimation filter. `num` is a vector containing the coefficients of the transposed FIR lowpass filter used for decimation. If omitted, a lowpass Nyquist filter with gain of 1 and cutoff frequency of π/m is the default.

Make this filter a fixed-point or single-precision filter by changing the value of the `Arithmetic` property for the filter `hm` as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating `hm`.

Input Argument	Description
num	Vector containing the coefficients of the FIR lowpass filter used for interpolation. When num is not provided as an input, firtdecim uses a lowpass Nyquist filter with gain equal to 1 and cutoff frequency equal to π/m by default. The default length for the Nyquist filter is 24*m. Therefore, each polyphase filter component has length 24.
m	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer. When you do not specify a value for m it defaults to 2.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating mfilt.firtdecim objects. The next table describes each property for an mfilt.firtdecim filter object.

Name	Values	Description
Arithmetic	Double, single, fixed	Specifies the arithmetic the filter uses to process data while filtering.
DecimationFactor	Integer	Decimation factor for the filter. m specifies the amount to reduce the sampling rate of the input signal. It must be an integer.

Name	Values	Description
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object. Also describes the signal flow for the filter object.
InputOffset	Integers	Contains a value derived from the number of input samples and the decimation factor — `InputOffset = mod(length(nx),m)` where `nx` is the number of input samples that have been processed so far and `m` is the decimation factor.
Numerator	Vector	Vector containing the coefficients of the FIR lowpass filter used for decimation.
PersistentMemory	[false], true	Determines whether the filter states get restored to zeros for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` set to `false` returns filter states to the default values after filtering. States that the filter does not change are not affected. Setting this to `true` allows you to modify the `States`, `InputOffset`, and `PolyphaseAccum` properties.

Name	Values	Description
PolyphaseAccum	Double, single [0]	The idea behind having both PolyphaseAccum and Accum is to differentiate between the adders in the filter that work in full precision at all times (PolyphaseAccum) from the adders in the filter that the user controls and that may introduce quantization effects when FilterInternals is set to SpecifyPrecision.
States	Double, single matching the filter arithmetic setting.	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the mfilt.firtdecim filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where hm is a filter.

For further information about the properties of this filter or any mfilt object, refer to "Multirate Filter Properties" on page 8-115.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits. [32]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.
AccumWordLength	Any integer number of bits [39]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to fixed allows you to modify other filter properties to customize your fixed-point filter.
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.

Name	Values	Description
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
OutputFracLength	Any positive or negative integer number of bits [32]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilterInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [39]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.

Name	Values	Description
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
PolyphaseAccum	fi object with zeros to start	Differentiates between the adders in the filter that work in full precision at all times (PolyphaseAccum) and the adders in the filter that the user controls and that may introduce quantization effects when FilterInternals is set to SpecifyPrecision.
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity.

Name	Values	Description
		• round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
		The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation. For information about the ordering of the states, refer to the filter structure section.

Filter Structure

To provide sample rate changes, mfilt.firtdecim uses the following structure. At the input you see a commutator that operates counterclockwise, moving from position 0 to position 2, position 1, and back to position 0 as input samples enter the filter. To keep track of the position of the commutator, the mfilt object uses the property InputOffset which reports the current position of the commutator in the filter.

The following figure details the signal flow for the direct form FIR filter implemented by mfilt.firtdecim.

mfilt.firtdecim
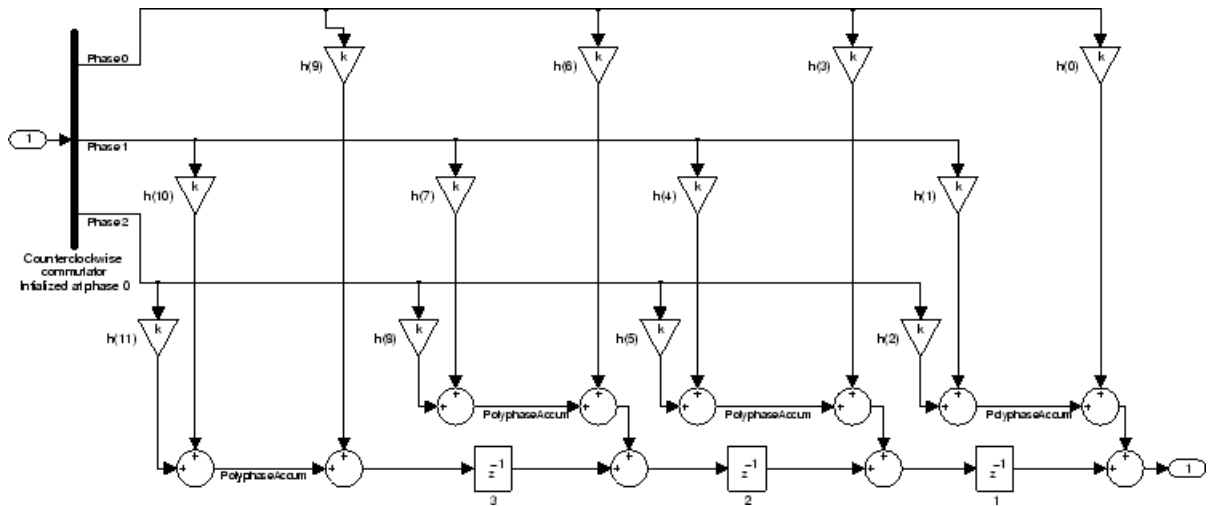
Notice the order of the states in the filter flow diagram. States 1 through 3 appear in the following diagram at each delay element. State 1 applies to the third delay element in phase 2. State 2 applies to the second delay element in phase 2. State 3 applies to the first delay element in phase 2. When you provide the states for the filter as a vector to the `States` property, the above description explains how the filter assigns the states you specify.

In property value form, the states for a filter `hm` are

```
hm.states=[1:3];
```

Examples   Demonstrate decimating an input signal by a factor of 2, in this case converting from 44.1 kHz down to 22.05 kHz. In the figure shown following the code, you see the results of decimating the signal.

```
m = 2;                  % Decimation factor.
hm = mfilt.firtdecim(m); % Use the default filter coeffs.
fs = 44.1e3;            % Original sample freq: 44.1 kHz.
n = 0:10239;            % 10240 samples, 0.232 second long signal
x  = sin(2*pi*1e3/fs*n); % Original signal--sinusoid at 1 kHz.
y = filter(hm,x);       % 5120 samples, 0.232 seconds.
```

```
stem(n(1:44)/fs,x(1:44)) % Plot original sampled at 44.1 kHz.
axis([0 0.001 -1.2 1.2]);
hold on                   % Plot decimated signal (22.05 kHz) in red
stem(n(1:22)/(fs/m),y(13:34),'r','filled')
xlabel('Time (sec)');ylabel('Signal Value');
legend('Original signal','Decimated signal','location','best');
```
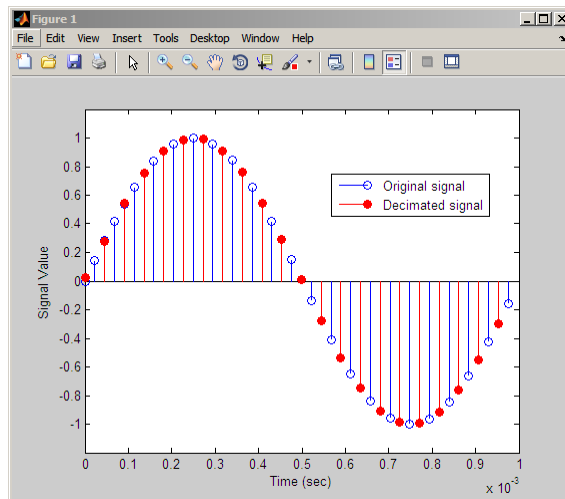

See Also   mfilt.firdecim | mfilt.firsrc | mfilt.cicdecim

mfilt.holdinterp

Purpose   FIR hold interpolator

Syntax   hm = mfilt.holdinterp(l)

Description   hm = mfilt.holdinterp(l) returns the object hm that represents a hold interpolator with the interpolation factor l. To work, l must be an integer. When you do not include l in the calling syntax, it defaults to 2. To perform interpolation by noninteger amounts, use one of the fractional interpolator objects, such as mfilt.firsrc.

When you use this hold interpolator, each sample added to the input signal between existing samples has the value of the most recent sample from the original signal. Thus you see something like a staircase profile where the interpolated samples form a plateau between the previous and next original samples. The example demonstrates this profile clearly. Compare this to the interpolation process for other interpolators in the toolbox, such as mfilt.linearinterp.

Make this filter a fixed-point or single-precision filter by changing the value of the Arithmetic property for the filter hm as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input arguments for creating hm.

Input Argument	Description
l	Interpolation factor for the filter. l specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for l it defaults to 2.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.holdinterp` objects. The next table describes each property for an `mfilt.interp` filter object.

Name	Values	Description
Arithmetic	Double, single, fixed	Specifies the arithmetic the filter uses to process data while filtering.
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of `mfilt` object.
Interpolation-Factor	Integer	Interpolation factor for the filter. `l` specifies the amount to increase the input sampling rate. It must be an integer.

mfilt.holdinterp

Name	Values	Description
PersistentMemory	'false' or 'true'	Determines whether the filter states are restored to zero for each filtering operation.
States	Double or single array	Filter states. states defaults to a vector of zeros that has length equal to nstates (hm). Always available, but visible in the display only when PersistentMemory is true.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the mfilt.holdinterp filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

info(hm)

where hm is a filter.

For further information about the properties of this filter or any mfilt object, refer to "Multirate Filter Properties" on page 8-115.

Name	Values	Description
Arithmetic	Double, single, fixed	Specifies the arithmetic the filter uses to process data while filtering.
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
Interpolation-Factor	Integer	Interpolation factor for the filter. l specifies the amount to increase the input sampling rate. It must be an integer.
PersistentMemory	'false' or 'true'	Determine whether the filter states get restored to zero for each filtering operation
States	fi object	Contains the filter states before, during, and after filter operations. For hold interpolators, the states are always empty — hold interpolators do not have states. The states use fi objects, with the

mfilt.holdinterp

Name	Values	Description
		associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolbox documentation.

Filter Structure

Hold interpolators do not have filter coefficients and their filter structure is trivial.

Examples

To see the effects of hold-based interpolation, interpolate an input sine wave from 22.05 to 44.1 kHz. Note that each added sample retains the value of the most recent original sample.

```
l = 2;                          % Interpolation factor
hm = mfilt.holdinterp(l);
fs = 22.05e3;                   % Original sample freq: 22.05 kHz.
n = 0:5119;                     % 5120 samples, 0.232 second long signal
x  = sin(2*pi*1e3/fs*n);        % Original signal, sinusoid at 1 kHz
y = filter(hm,x);               % 10240 samples, still 0.232 seconds
stem(n(1:22)/fs,x(1:22),'filled')  % Plot original sampled at
                                % 22.05 kHz
hold on                         % Plot interpolated signal (44.1 kHz)
stem(n(1:44)/(fs*l),y(1:44),'r')
legend('Original Signal','Interpolated Signal','Location','best');
xlabel('Time (sec)');ylabel('Signal Value')
```

The following figure shows clearly the step nature of the signal that comes from interpolating the signal using the hold algorithm approach. Compare the output to the linear interpolation used in `mfilt.linearinterp`.
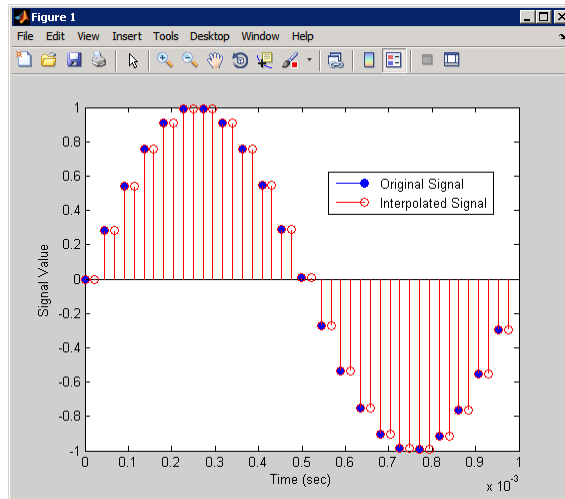
See Also   mfilt.linearinterp | mfilt.firinterp | mfilt.firsrc | mfilt.cicinterp

mfilt.iirdecim

Purpose   IIR decimator

Syntax   `hm = mfilt.iirdecim(c1,c2,...)`

Description   `hm = mfilt.iirdecim(c1,c2,...)` constructs an IIR decimator filter given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The resulting IIR decimator is a polyphase IIR filter where each phase is a cascade allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadeallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirdecim` interprets them same way as `mfilt.cascadeallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadeallpass` section. This exception case occurs with quasi-linear phase IIR decimators.

Although the first example shows how to construct an IIR decimators explicitly, one usually constructs an IIR decimators filter as a result of designing an decimators, as shown in the subsequent examples.

Examples   When the coefficients are known, you can construct the IIR decimator directly using `mfilt.iirdecim`. For example, if the filter's coefficients are [0.6 0.5] for the first phase in the first stage, 0.7 for the second phase in the first stage and 0.8 for the third phase in the first stage; as well as 0.5 for the first phase in the second stage and 0.4 for the second phase in the second stage, construct the filter as shown here.

`Hm = mfilt.iirdecim({[0.6 0.5] 0.7 0.8},{0.5 0.4})`

Also refer to the "Quasi-Linear Phase Halfband and Dyadic Halfband Designs" section of the "IIR Polyphase Filter Design" example, iirallpassdemo example.

When the coefficients are not known, use the approach given by the following set of examples. Start by designing an elliptic halfband decimator with a decimation factor of 2. The example specifies the optional sampling frequency argument.

```
tw = 100;   % Transition width of filter.
ast = 80;   % Stopband attenuation of filter.
fs = 2000;  % Sampling frequency of signal to filter.
m = 2;      % Decimation factor.
hm = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

hm contains the specifications for a decimator defined by tw, ast, m, and fs.

Use the specification object hm to design a mfilt.iirdecim filter object.

```
d = design(hm,'ellip','filterstructure','iirdecim');
% Note that realizemdl requires Simulink
realizemdl(d)       % Build model of the filter.
```

Designing a linear phase decimator is similar to the previous example. In this case, design a halfband linear phase decimator with decimation factor of 2.

```
tw  = 100;  % Transition width of filter.
ast = 60;   % Stopband attenuation of filter.
fs  = 2000; % Sampling frequency of signal to filter.
m = 2;      % Decimation factor.
```

Create a specification object for the decimator.

```
hm = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

Finally, design the filter d.

```
d = design(hm,'iirlinphase','filterstructure','iirdecim');
```

```
% Note that realizemdl requires Simulink
realizemdl(d)  % Build model of the filter.
```

The filter implementation appears in this model, generated by `realizemdl` and Simulink.

Given the design specifications shown here

the first phase is a delay section with 0s and a 1 for coefficients and the second phase is a linear phase decimator, shown in the next models.
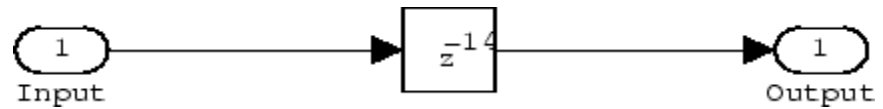
Phase 1 model
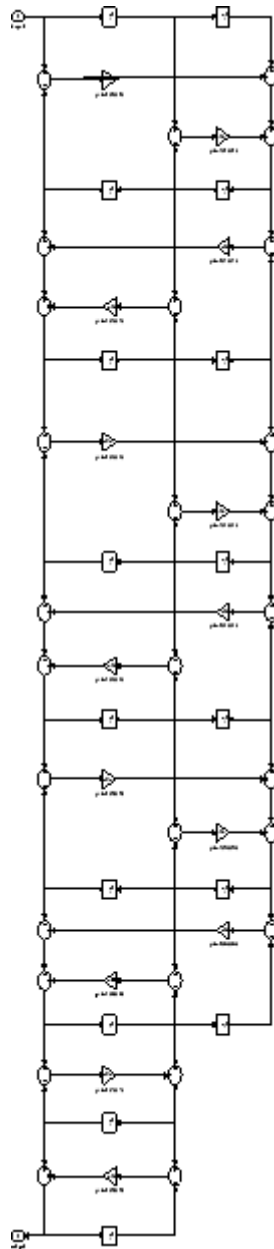
Phase 2 model
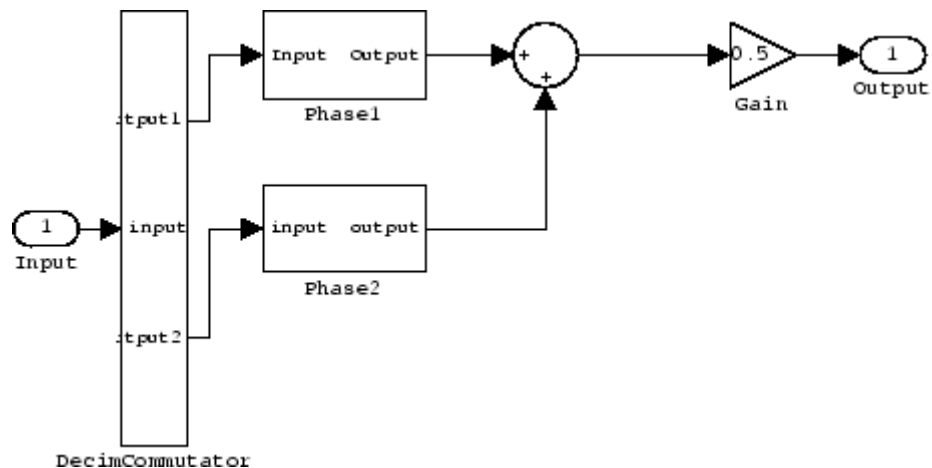
Overall model

For more information about Multirate Filter Constructors see the "Getting Started with Multirate Filter (MFILT) Objects" example, mfiltgettingstarteddemo.

See Also   dfilt.cascadeallpass | mfilt | mfilt.iirinterp | mfilt.iirwdfdecim

Purpose   IIR interpolator

Syntax   hm = mfilt.iirinterp(c1,c2,...)

Description   hm = mfilt.iirinterp(c1,c2,...) constructs an IIR interpolator filter given the coefficients specified in the cell arrays C1, C2, etc.

The IIR interpolator is a polyphase IIR filter where each phase is a cascade allpass IIR filter.

Each cell array ci contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to dfilt.cascadeallpass. The contents of the cell arrays are the same for both filter constructors and mfilt.iirdecim interprets them same way as mfilt.cascadeallpass.

The following exception applies to interpreting the contents of a cell array—if one of the cell arrays ci contains only one vector, and that vector comprises a series of 0s and a unique element equal to 1, that cell array represents a dfilt.delay section with latency equal to the number of zeros, rather than a dfilt.cascadeallpass section. This exception case occurs with quasi-linear phase IIR interpolators.

Although the first example shows how to construct an IIR interpolator explicitly, one usually constructs an IIR interpolator filter as a result of designing an interpolator, as shown in the subsequent examples.

Examples   When the coefficients are known, you can construct the IIR interpolator directly using mfilt.iirinterp. In the following example, a cascaded polyphase IIR interpolator filter is constructed using 2 phases for each of three stages. The coefficients are given below:

```
Phase1Sect1=0.0603;Phase1Sect2=0.4126; Phase1Sect3=0.7727;
Phase2Sect1=0.2160; Phase2Sect2=0.6044; Phase2Sect3=0.9239;
```

Next the filter is implemented by passing the above coefficients to `mfilt.iirinterp` as cell arrays, where each cell array represents a different phase.

```
Hm = mfilt.iirinterp({Phase1Sect1,Phase1Sect2,Phase1Sect3},...
{Phase2Sect1,Phase2Sect2,Phase2Sect3})


Hm =

        FilterStructure: 'IIR Polyphase Interpolator'
              Polyphase: Phase1: Section1: 0.0603
                                 Section2: 0.4126
                                 Section3: 0.7727
                         Phase2: Section1: 0.216
                                 Section2: 0.6044
                                 Section3: 0.9239

   InterpolationFactor: 2
      PersistentMemory: false
```

Also refer to the "Quasi-Linear Phase Halfband and Dyadic Halfband Designs" section of the "IIR Polyphase Filter Design" example, iirallpassdemo example.

When the coefficients are not known, use the approach given by the following set of examples. Start by designing an elliptic halfband interpolator with a interpolation factor of 2.

```
tw = 100;  % Transition width of filter.
ast = 80;   % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of filter.
l  = 2;    % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

Specification object d stores the interpolator design specifics. With the details in d, design the filter, returning hm, an `mfilt.iirinterp` object. Use hm to realize the filter if you have Simulink installed.

```
hm = design(d,'ellip','filterstructure','iirinterp');
```

```
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

Designing a linear phase halfband interpolator follows the same pattern.

```
tw = 100;  % Transition width of filter.
ast= 60;   % Stopband attenuation of filter.
fs = 2000; % Sampling frequency of filter.
l = 2;     % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

`fdesign.interpolator` returns a specification object that stores the design features for an interpolator.

Now perform the actual design that results in an `mfilt.iirinterp` filter, `hm`.

```
hm = design(d,'iirlinphase','filterstructure','iirinterp');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

The toolbox creates a Simulink model for `hm`, shown here. Phase1, Phase2, and InterpCommutator are all subsystem blocks.
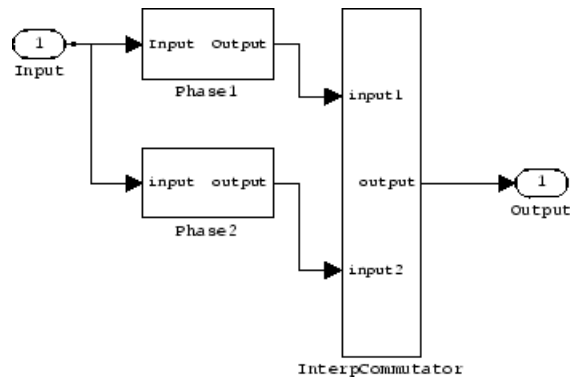
For more information about Multirate Filter Constructors see the "Getting Started with Multirate Filter (MFILT) Objects" example, mfiltgettingstarteddemo.

See Also   dfilt.cascadeallpass | mfilt | mfilt.iirdecim | mfilt.iirwdfinterp

Purpose   IIR wave digital filter decimator

Syntax   hm = mfilt.iirwdfdecim(c1,c2,...)

Description   hm = mfilt.iirwdfdecim(c1,c2,...) constructs an IIR wave digital decimator given the coefficients specified in the cell arrays c1, c2, and so on. The IIR decimator hm is a polyphase IIR filter where each phase is a cascade wave digital allpass IIR filter.

Each cell array ci contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to dfilt.cascadewdfallpass. The contents of the cell arrays are the same for both filter constructors and mfilt.iirwdfdecim interprets them same way as mfilt.cascadewdfallpass.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays ci contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a dfilt.delay section with latency equal to the number of zeros, rather than a dfilt.cascadewdfallpass section. This exception occurs with quasi-linear phase IIR decimators.

Usually you do not construct IIR wave digital filter decimators explicitly. Instead, you obtain an IIR wave digital filter decimator as a result of designing a halfband decimator. The first example in the following section illustrates this case.

Examples   Design an elliptic halfband decimator with a decimation factor equal to 2. Both examples use the iirwdfdecim filter structure (an input argument to the design method) to design the final decimator.

The first portion of this example generates a filter specification object d that stores the specifications for the decimator.

```
tw  = 100; % Transition width of filter to design, 100 Hz.
ast = 80;  % Stopband attenuation of filter 80 dB.
fs  = 2000; % Sampling frequency of the input signal.
m   = 2;   % Decimation factor.
```

```
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

Now perform the actual design using d. Filter object hm is an
mfilt.iirwdfdecim filter.

```
Hm = design(d,'ellip','FilterStructure','iirwdfdecim');
% Note that realizemdl requires Simulink
realizemdl(hm)      % Build model of the filter.
```

Design a linear phase halfband decimator for decimating a signal by a
factor of 2.

```
tw  = 100;  % Transition width of filter, 100 Hz.
ast = 60;   % Filter stopband attenuation = 80 dB
fs  = 2000; % Input signal sampling frequency.
m   = 2;    % Decimation factor.
d = fdesign.decimator(m,'halfband','tw,ast',tw,ast,fs);
```

Use d to design the final filter hm, an mfilt.iirwdfdecim object.

```
hm = design(d,'iirlinphase','filterstructure',...
'iirwdfdecim');
% Note that realizemdl requires Simulink
realizemdl(hm)      % Build model of the filter.
```

The models that `realizemdl` returns for each example appear below. At this level, the realizations of the filters are identical. The differences appear in the subsystem blocks Phase1 and Phase2.
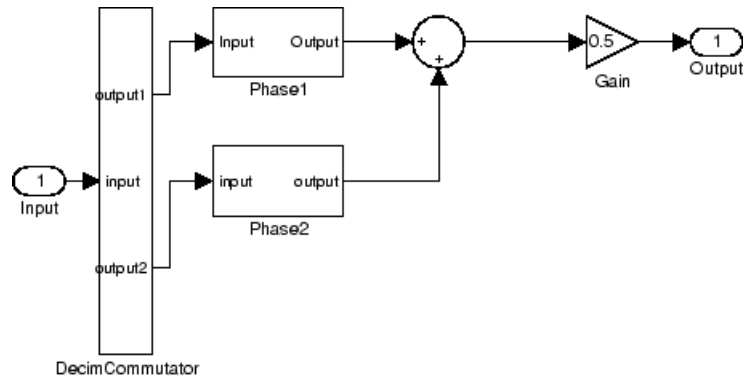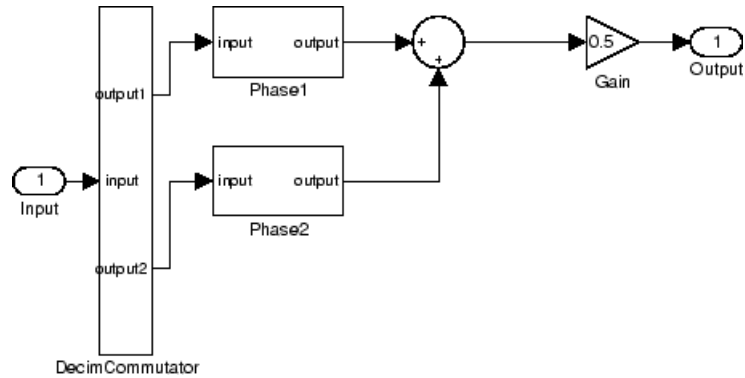
This is the Phase1 subsystem from the halfband model.
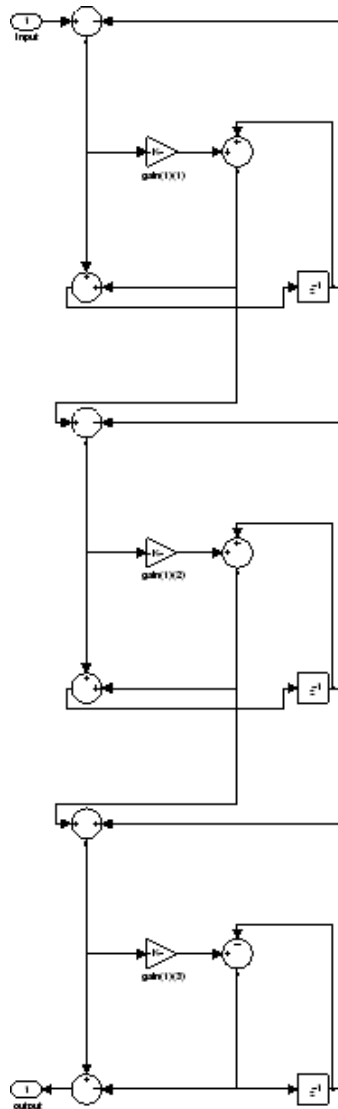
Phase1 subsystem from the linear phase model is less revealing—an allpass filter.
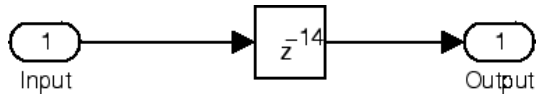
See Also dfilt.cascadewdfallpass | mfilt | mfilt.iirdecim | mfilt.iirwdfinterp

mfilt.iirwdfinterp

Purpose　　IIR wave digital filter interpolator

Syntax　　`hm = mfilt.iirwdfinterp(c1,c2,...)`

Description　　`hm = mfilt.iirwdfinterp(c1,c2,...)` constructs an IIR wave digital interpolator given the coefficients specified in the cell arrays `c1`, `c2`, and so on. The IIR interpolator `hm` is a polyphase IIR filter where each phase is a cascade wave digital allpass IIR filter.

Each cell array `ci` contains a set of vectors representing a cascade of allpass sections. Each element in one cell array is one section. For more information about the contents of each cell array, refer to `dfilt.cascadewdfallpass`. The contents of the cell arrays are the same for both filter constructors and `mfilt.iirwdfinterp` interprets them same way as `mfilt.cascadewdfallpass`.

The following exception applies to interpreting the contents of a cell array — if one of the cell arrays `ci` contains only one vector, and that vector comprises a series of 0s and one element equal to 1, that cell array represents a `dfilt.delay` section with latency equal to the number of zeros, rather than a `dfilt.cascadewdfallpass` section. This exception occurs with quasi-linear phase IIR interpolators.

Usually you do not construct IIR wave digital filter interpolators explicitly. Rather, you obtain an IIR wave digital interpolator as a result of designing a halfband interpolator. The first example in the following section illustrates this case.

Examples　　Design an elliptic halfband interpolator with interpolation factor equal to 2. At the end of the design process, hm is an IIR wave digital filter interpolator.

```
tw  = 100;  % Transition width of filter, 100 Hz.
ast = 80;   % Stopband attenuation of filter, 80 dB.
fs  = 2000; % Sampling frequency after interpolation.
l   = 2;    % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

The specification object d stores the interpolator design requirements. Now use d to design the actual filter hm.

```
hm = design(d,'ellip','filterstructure','iirwdfinterp');
```

If you have Simulink installed, you can realize your filter as a model built from DSP System Toolbox blocks.

```
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

For variety, design a linear phase halfband interpolator with an interpolation factor of 2.

```
tw  = 100;  % Transition width of filter, 100 Hz.
ast = 80;   % Stopband attenuation of filter, 80 dB.
fs  = 2000; % Sampling frequency after interpolation.
l   = 2;    % Interpolation factor.
d = fdesign.interpolator(l,'halfband','tw,ast',tw,ast,fs);
```

Now perform the actual design process with d. Filter hm is an IIR wave digital filter interpolator. As in the previous example, realizemdl returns a Simulink model of the filter if you have Simulink installed.

```
hm = design(d,'iirlinphase','filterstructure',...
'iirwdfinterp');
% Note that realizemdl requires Simulink
realizemdl(hm)     % Build model of the filter.
```

See Also   dfilt.cascadewdfallpass | mfilt.iirinterp | mfilt.iirwdfdecim

mfilt.linearinterp

Purpose	Linear interpolator

Syntax   hm = mfilt.linearinterp(l)

Description   hm = mfilt.linearinterp(l) returns an FIR linear interpolator hm
with an integer interpolation factor l. Provide l as a positive integer.
The default value for the interpolation factor is 2 when you do not
include the input argument l.

When you use this linear interpolator, the samples added to the input
signal have values between the values of adjacent samples in the
original signal. Thus you see something like a smooth profile where the
interpolated samples continue a line between the previous and next
original samples. The example demonstrates this smooth profile clearly.
Compare this to the interpolation process for mfilt.holdinterp, which
creates a stairstep profile.

Make this filter a fixed-point or single-precision filter by changing the
value of the Arithmetic property for the filter hm as follows:

- To change to single-precision filtering, enter

  ```
  set(hm,'arithmetic','single');
  ```

- To change to fixed-point filtering, enter

  ```
  set(hm,'arithmetic','fixed');
  ```

Input Arguments

The following table describes the input argument for
mfilt.linearinterp.

Input Argument	Description
l	Interpolation factor for the filter. l specifies the amount to increase the input sampling rate. It must be an integer. When you do not specify a value for l it defaults to 2.

Object Properties

This section describes the properties for both floating-point filters (double-precision and single-precision) and fixed-point filters.

Floating-Point Filter Properties

Every multirate filter object has properties that govern the way it behaves when you use it. Note that many of the properties are also input arguments for creating `mfilt.linearinterp` objects. The next table describes each property for an `mfilt.linearinterp` filter object.

Name	Values	Description
Arithmetic	Double, single, fixed	Specifies the arithmetic the filter uses to process data while filtering.
FilterStructure	String	Reports the type of filter object. You cannot set this property — it is always read only and results from your choice of mfilt object.
InterpolationFactor	Integer	Interpolation factor for the filter. l specifies the amount to increase the input sampling rate. It must be an integer.
PersistentMemory	'false' or 'true'	Determine whether the filter states get restored to zero for each filtering operation
States	Double or single array	Filter states. states defaults to a vector of zeros that has length equal to nstates(hm). Always available, but visible in the display only when PersistentMemory is true.

Fixed-Point Filter Properties

This table shows the properties associated with the fixed-point implementation of the `mfilt.holdinterp` filter.

Note The table lists all of the properties that a fixed-point filter can have. Many of the properties listed are dynamic, meaning they exist only in response to the settings of other properties. To view all of the characteristics for a filter at any time, use

```
info(hm)
```

where `hm` is a filter.

For further information about the properties of this filter or any `mfilt` object, refer to "Multirate Filter Properties" on page 8-115.

Name	Values	Description
AccumFracLength	Any positive or negative integer number of bits. Depends on L. [29 when L=2]	Specifies the fraction length used to interpret data output by the accumulator.
AccumWordLength	Any integer number of bits [33]	Sets the word length used to store data in the accumulator.
Arithmetic	fixed for fixed-point filters	Setting this to `fixed` allows you to modify other filter properties to customize your fixed-point filter.

Name	Values	Description
CoeffAutoScale	[true], false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength property value to specify the precision used.
CoeffWordLength	Any integer number of bits [16]	Specifies the word length to apply to filter coefficients.
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter automatically sets the output word and fraction lengths, product word and fraction lengths, and the accumulator word and fraction lengths to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision makes the output and accumulator-related properties available so you can set your own word and fraction lengths for them.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret input data.
InputWordLength	Any integer number of bits [16]	Specifies the word length applied to interpret input data.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.

Name	Values	Description
OutputFracLength	Any positive or negative integer number of bits [29]	Determines how the filter interprets the filter output data. You can change the value of OutputFracLength when you set FilterInternals to SpecifyPrecision.
OutputWordLength	Any integer number of bits [33]	Determines the word length used for the output data. You make this property editable by setting FilterInternals to SpecifyPrecision.
OverflowMode	saturate, [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic.) The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.

Name	Values	Description
RoundMode	[convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

Name	Values	Description
Signed	[true], false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
States	fi object	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. The states use fi objects, with the associated properties from those objects. For details, refer to fixed-point objects in Fixed-Point Toolboxdocumentation. For information about the ordering of the states, refer to the filter structure in the following section.

Filter Structure

Linear interpolator structures depend on the FIR filter you use to implement the filter. By default, the structure is direct-form FIR.
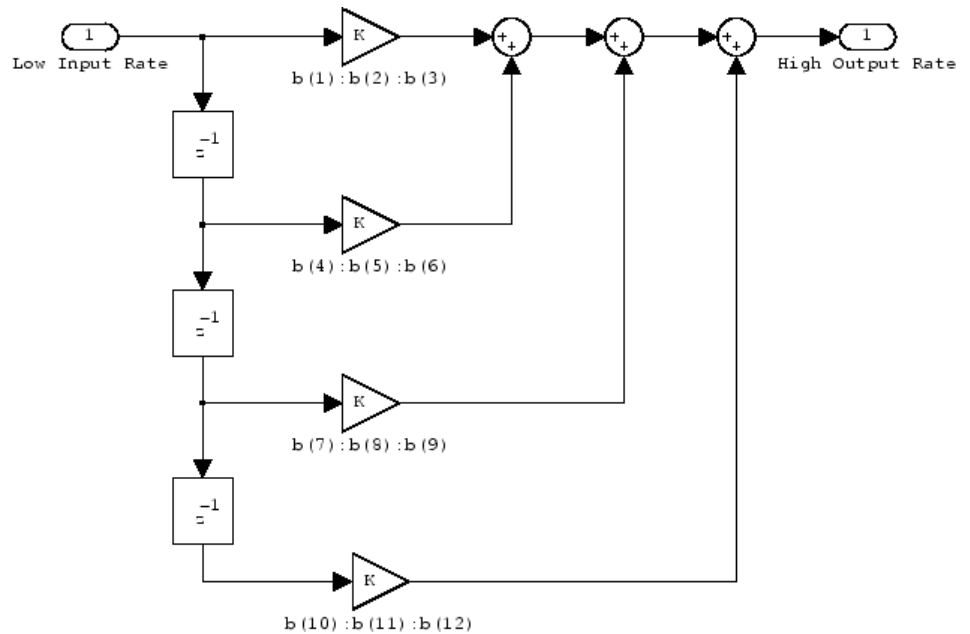
Examples

Interpolation by a factor of 2 (used to convert the input signal sampling rate from 22.05 kHz to 44.1 kHz).

```
l = 2;                    % Interpolation factor
hm = mfilt.linearinterp(l);
fs = 22.05e3;             % Original sample freq: 22.05 kHz.
n = 0:5119;               % 5120 samples, 0.232 second long signal
x  = sin(2*pi*1e3/fs*n);  % Original signal, sinusoid at 1 kHz
y = filter(hm,x);         % 10240 samples, still 0.232 seconds
stem(n(1:22)/fs,x(1:22),'filled') % Plot original sampled at
                                  % 22.05 kHz
hold on                   % Plot interpolated signal (44.1
                          % kHz) in red
```

```
stem(n(1:44)/(fs*l),y(2:45),'r')
xlabel('Time (s)');ylabel('Signal Value')
```

Using linear interpolation, as compared to the hold approach of `mfilt.holdinterp`, provides greater fidelity to the original signal.
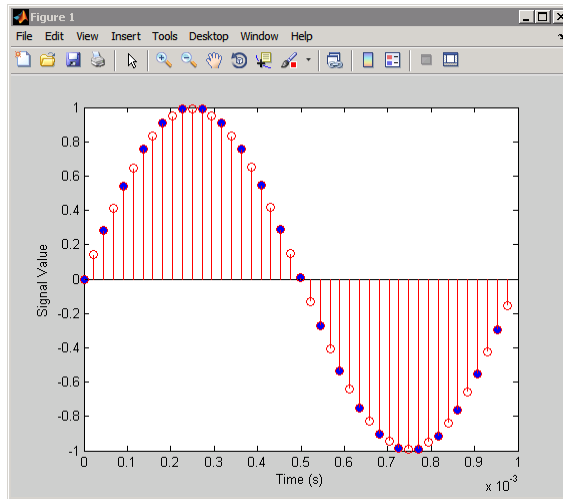
See Also   `mfilt.holdinterp` | `mfilt.firinterp` | `mfilt.firsrc` | `mfilt.cicinterp`

minimizecoeffwl

Purpose   Minimum wordlength fixed-point filter

Syntax   Hq = minimizecoeffwl(Hd)
                 Hq = minimizecoeffwl(Hd,...,'NoiseShaping',NSFlag)
                 Hq = inimizecoeffwl(Hd,...,'NTrials',N)
                 Hq = minimizecoeffwl(Hd,...'Apasstol',Apasstol)
                 Hq = minimizecoeffwl(Hd,...,'Astoptol',Astoptol)
                 Hq = minimizecoeffwl(Hd,...,'MatchrefFilter',RefFiltFlag)

Description Hq = minimizecoeffwl(Hd) returns the minimum wordlength
                 fixed-point filter object Hq that meets the design specifications of the
                 single-stage or multistage FIR filter object Hd. Hd must be generated
                 using fdesign and design. If Hd is a multistage filter object, the
                 procedure minimizes the wordlength for each stage separately.
                 minimizecoeffwl uses a stochastic noise-shaping procedure by default
                 to minimize the wordlength. To obtain repeatable results on successive
                 function calls, initialize the uniform random number generator rand.

                 Hq = minimizecoeffwl(Hd,...,'NoiseShaping',NSFlag) enables or
                 disables the stochastic noise-shaping procedure in the minimization of
                 the wordlength. By default NSFlag is true. Setting NSFlag to false
                 minimizes the wordlength without using noise-shaping.

                 Hq = inimizecoeffwl(Hd,...,'NTrials',N) specifies the number of
                 Monte Carlo trials to use in the minimization. Hq is the filter with the
                 minimum wordlength among the N trials that meets the specifications
                 in Hd. 'NTrials' defaults to one.

                 Hq = minimizecoeffwl(Hd,...'Apasstol',Apasstol) specifies the
                 passband ripple tolerance in dB. 'Apasstol' defaults to 1e-4.

                 Hq = minimizecoeffwl(Hd,...,'Astoptol',Astoptol) specifies the
                 stopband tolerance in dB. 'Astoptol' defaults to 1e-2.

                 Hq = minimizecoeffwl(Hd,...,'MatchrefFilter',RefFiltFlag)
                 determines whether the fixed-point filter matches the filter order and
                 transition width of the floating-point design. Setting 'MatchRefFilter'
                 to true returns a fixed-point filter with the same order and transition
                 width as Hd. The 'MatchRefFilter' property defaults to false and the

resulting fixed-point filter may have a different order and transition width than the floating-point design Hd.

You must have the Fixed-Point Toolbox software installed to use this function.

Examples   Minimize wordlength for lowpass FIR equiripple filter:

```
f=fdesign.lowpass('Fp,Fst,Ap,Ast',0.2,0.3,1,60);
% Design filter with double-precision floating point
Hd=design(f,'equiripple');
% Find minimum wordlength fixed-point filter
% with 0.15 dB stopband tolerance
Hq=minimizecoeffwl(Hd,'MatchRefFilter',true,'Astoptol',0.15);
Hq1=minimizecoeffwl(Hd,'Astoptol',0.15);
% Hq.coeffwordlength is 14 bits.
% Hq1.coeffwordlength is 11 bits
hfvt=fvtool(Hd,Hq,Hq1,'showreference','off');
legend(hfvt,'44 coefficients floating-point',...
'44 coefficients 14 bits','48 coefficients 11 bits');
```
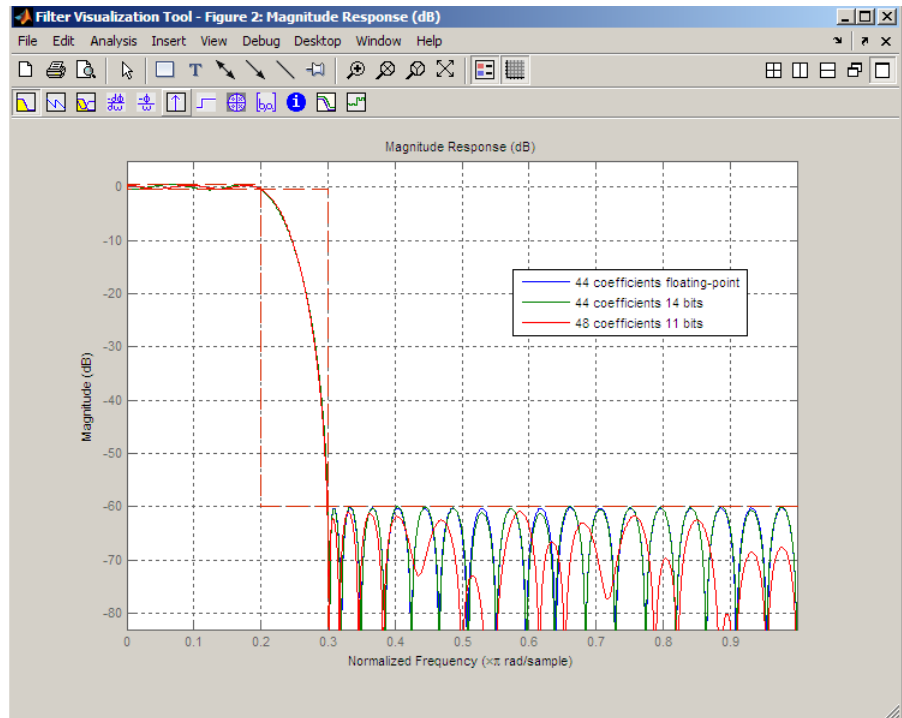
See Also   constraincoeffwl | design | fdesign | maximizestopband | measure | rand

Tutorials   • "Fixed-Point Data Types"

msepred

Purpose   Predicted mean-squared error for adaptive filter

Syntax
```
[mmse,emse] = msepred(ha,x,d)
[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d)
[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d,m)
```

Description   `[mmse,emse] = msepred(ha,x,d)` predicts the steady-state values at convergence of the minimum mean-squared error (`mmse`) and the excess mean-squared error (`emse`) given the input and desired response signal sequences in `x` and `d` and the property values in the adaptfilt object `ha`.

`[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d)` calculates three sequences corresponding to the analytical behavior of the LMS adaptive filter defined by `ha`:

- `meanw` — contains the sequence of coefficient vector means. The columns of matrix `meanw` contain predictions of the mean values of the LMS adaptive filter coefficients at each time instant. The dimensions of `meanw` are `(size(x,1))`-by-`(ha.length)`.

- `mse` — contains the sequence of mean-square errors. This column vector contains predictions of the mean-square error of the LMS adaptive filter at each time instant. The length of `mse` is equal to `size(x,1)`.

- `tracek` — contains the sequence of total coefficient error powers. This column vector contains predictions of the total coefficient error power of the LMS adaptive filter at each time instant. The length of `tracek` is equal to `size(x,1)`.

`[mmse,emse,meanw,mse,tracek] = msepred(ha,x,d,m)` specifies an optional input argument `m` that is the decimation factor for computing `meanw`, `mse`, and `tracek`. When `m > 1`, `msepred` saves every `m`th predicted value of each of these sequences. When you omit the optional argument `m`, it defaults to one.

Note msepred is available for the following adaptive filters only:
— adaptfilt.blms — adaptfilt.blmsfft — adaptfilt.lms —
adaptfilt.nlms — adaptfilt.se Using msepred is the same for any
adaptfilt object constructed by the supported filters.

Examples   Analyze and simulate a 32-coefficient adaptive filter using 25 trials of
2000 iterations each.

```
x = zeros(2000,25); d = x;     % Initialize variables
ha = fir1(31,0.5);             % FIR system to be identified
x = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x))));
n = 0.1*randn(size(x));        % observation noise signal
d = filter(ha,1,x)+n;          % desired signal
l = 32;                        % Filter length
mu = 0.008;                    % LMS step size.
m = 5;                         % Decimation factor for analysis
                               % and simulation results
ha = adaptfilt.lms(l,mu);
[mmse,emse,meanW,mse,traceK] = msepred(ha,x,d,m);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
nn = m:m:size(x,1);
subplot(2,1,1);
plot(nn,meanWsim(:,12),'b',nn,meanW(:,12),'r',nn,...
meanWsim(:,13:15),'b',nn,meanW(:,13:15),'r');
PlotTitle ={'Average Coefficient Trajectories for';...
            'W(12), W(13), W(14), and W(15)'};
title(PlotTitle);
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Coefficient Value');
subplot(2,2,3);
semilogy(nn,simmse,[0 size(x,1)],[(emse+mmse)...
(emse+mmse)],nn,mse,[0 size(x,1)],[mmse mmse]);
title('Mean-Square Error Performance');
axis([0 size(x,1) 0.001 10]);
legend('MSE (Sim.)','Final MSE','MSE','Min. MSE');
```

```
xlabel('Time Index'); ylabel('Squared Error Value');
subplot(2,2,4);
semilogy(nn,traceKsim,nn,traceK,'r');
title('Sum-of-Squared Coefficient Errors'); axis([O size(x,1)...
0.0001 1]);
legend('Simulation','Theory');
xlabel('Time Index'); ylabel('Squared Error Value');
```

Viewing the plots in this figure you see the various error values plotted in both simulation and theory. Each subplot reveals more information about the results as the simulation converges with the theoretical performance.
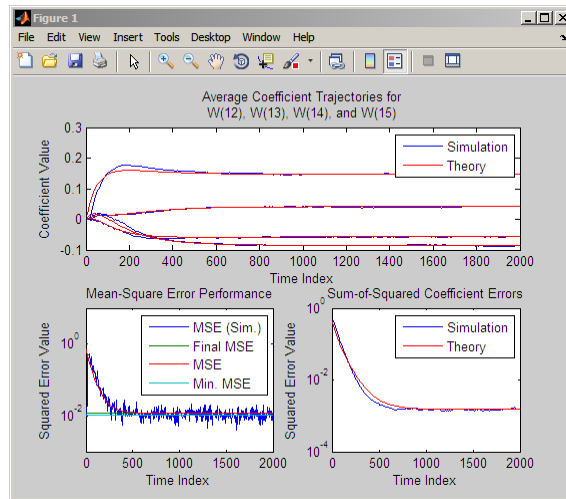
See Also   filter | maxstep | msesim

Purpose   Measured mean-squared error for adaptive filter

Syntax   mse = msesim(ha,x,d)
                   [mse,meanw,w,tracek] = msesim(ha,x,d)
                   [mse,meanw,w,tracek] = msesim(ha,x,d,m)

Description   mse = msesim(ha,x,d) returns the sequence of mean-square errors in
                   column vector mse. The vector contains estimates of the mean-square
                   error of the adaptive filter at each time instant during adaptation. The
                   length of mse is equal to size(x,1). The columns of matrix x contain
                   individual input signal sequences, and the columns of the matrix d
                   contain corresponding desired response signal sequences.

                   [mse,meanw,w,tracek] = msesim(ha,x,d) calculates three
                   parameters that correspond to the simulated behavior of the adaptive
                   filter defined by ha:

                   • meanw — sequence of coefficient vector means. The columns of this
                     matrix contain estimates of the mean values of the LMS adaptive
                     filter coefficients at each time instant. The dimensions of meanw are
                     (size(x,1))-by-(ha.length).

                   • w — estimate of the final values of the adaptive filter coefficients for
                     the algorithm corresponding to ha.

                   • tracek — sequence of total coefficient error powers. This column
                     vector contains estimates of the total coefficient error power of the
                     LMS adaptive filter at each time instant. The length of tracek is
                     equal to size(X,1).

                   [mse,meanw,w,tracek] = msesim(ha,x,d,m) specifies an optional
                   input argument m that is the decimation factor for computing meanw,
                   mse, and tracek. When m > 1, msepsim saves every mth predicted value
                   of each of these sequences. When you omit the optional argument m, it
                   defaults to one.

Examples   Simulation of a 32-coefficient FIR filter using 25 trials, each trial
                   having 2000 iterations of the adaptation process.

```
x = zeros(2000,25); d = x;          % Initialize variables
ha = fir1(31,0.5);                  % FIR system to be identified
x = filter(sqrt(0.75),[1 -0.5],sign(randn(size(x))));
n = 0.1*randn(size(x));             % Observation noise signal
d = filter(ha,1,x)+n;               % Desired signal
l = 32;                             % Filter length
mu = 0.008;                         % LMS Step size.
m  = 5;                             % Decimation factor for analysis
                                    % and simulation results
ha = adaptfilt.lms(l,mu);
[simmse,meanWsim,Wsim,traceKsim] = msesim(ha,x,d,m);
nn = m:m:size(x,1);
subplot(2,1,1);
plot(nn,meanWsim(:,12),'b',nn,meanWsim(:,13:15),'b');
PlotTitle ={'Average Coefficient Trajectories for';...
            'W(12), W(13), W(14), and W(15)'};
title(PlotTitle);
xlabel('Time Index'); ylabel('Coefficient Value');
subplot(2,2,3);
semilogy(nn,simmse);
title('Mean-Square Error Performance'); axis([0 size(x,1) 0.001...
10]);
legend('Measured MSE');
xlabel('Time Index'); ylabel('Squared Error Value');
subplot(2,2,4);
semilogy(nn,traceKsim);
title('Sum-of-Squared Coefficient Errors'); axis([0 size(x,1)...
0.0001 1]);
xlabel('Time Index'); ylabel('Squared Error Value');
```

Calculating the mean squared error for an adaptive filter is one measure
of the performance of the adapting algorithm. In this figure, you see a
variety of measures of the filter, including the error values.
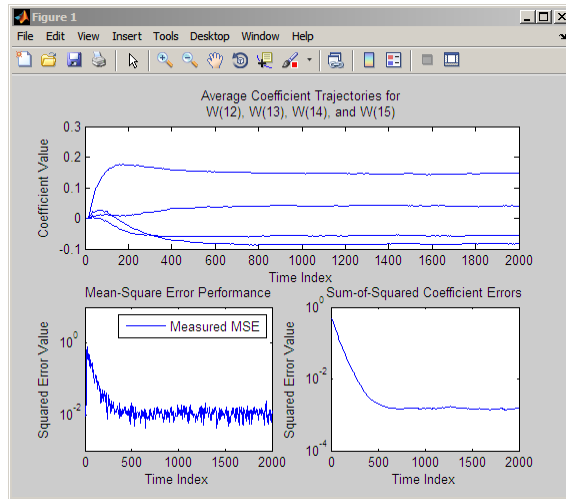
See Also   filter | msepred

multistage

Purpose   Multistage filter from specification object

Syntax
```
hd = design(d,'multistage')
hd = design(...,'filterstructure',structure)
hd = design(...,'nstages',nstages)
hd = design(...,'usehalfbands',hb)
```

Description   `hd = design(d,'multistage')` designs a multistage filter whose response you specified by the filter specification object `d`.

`hd = design(...,'filterstructure',structure)` returns a filter with the structure specified by `structure`. Input argument `structure` is `dffir` by default and can also be one of the following strings.

structure String	Valid with These Responses
firdecim	Lowpass or Nyquist response
firtdecim	Lowpass or Nyquist response
firinterp	Lowpass or Nyquist response
lowpass	Default lowpass only

Multistage design applies to the default lowpass filter specification object and to decimators and interpolators that use either lowpass or Nyquist responses.

`hd = design(...,'nstages',nstages)` specifies `nstages`, the number of stages to be used in the design. `nstages` must be an integer or the string `auto`. To allow the design algorithm to use the optimal number of stages while minimizing the cost of using the resulting filter, `nstages` is `auto` by default. When you specify an integer for `nstages`, the design algorithm minimizes the cost for the number of stages you specify.

`hd = design(...,'usehalfbands',hb)` uses halfband filters when you set `hb` to `true`. The default value for `hb` is `false`.

Note To see a list of the design methods available for your filter, use designmethods(hd).

Examples

Design a minimum-order, multistage Nyquist interpolator.

```
l = 15;   % Interpolation factor. Also the Nyquist band.
tw = 0.05; % Normalized transition width
ast = 40;   % Minimum stopband attenuation in dB
d = fdesign.interpolator(l,'nyquist',l,'tw,ast',tw,ast);
hm = design(d,'multistage');
fvtool(hm);
```
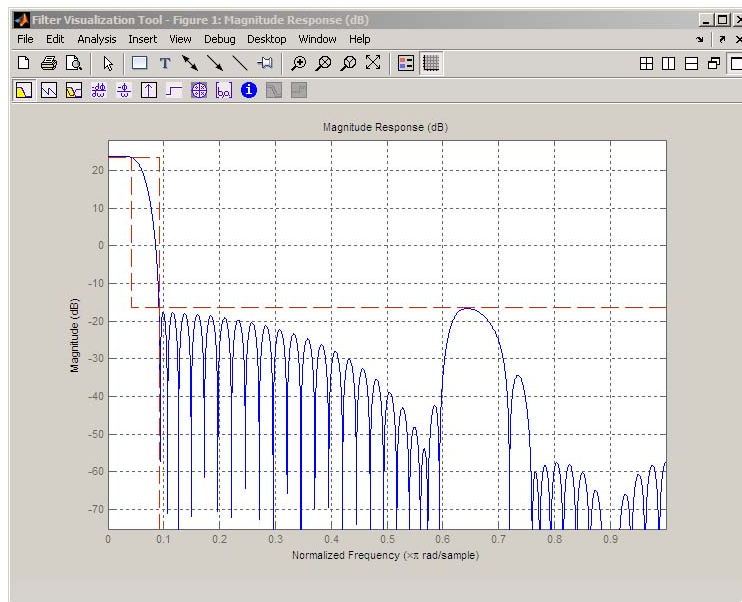

Design a multistage lowpass interpolator with an interpolation factor of 8.

```
m = 8; % Interpolation factor;
```

```
d = fdesign.interpolator(m,'lowpass');
% Use halfband filters if possible.
hm = design(d,'multistage','Usehalfbands',true);
fvtool(hm);
```

This figure shows the response for `hm`.
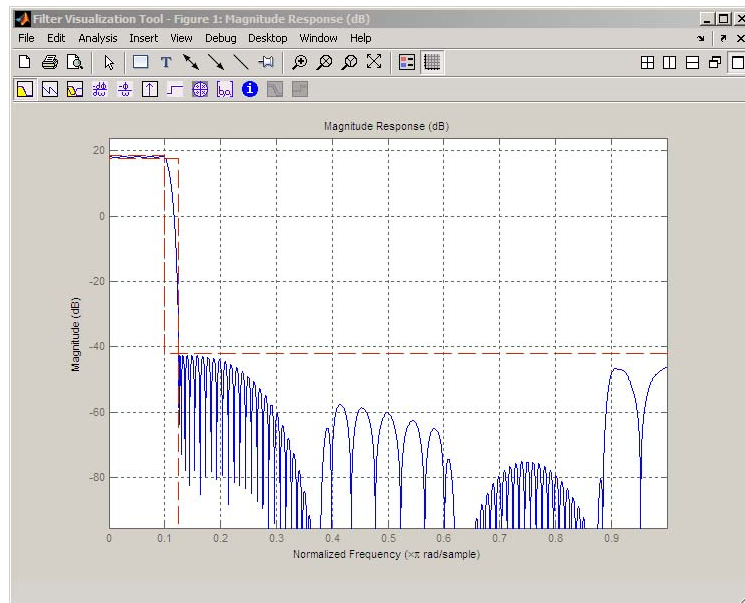
See Also   design | designopts

Purpose   Power spectral density of filter output due to roundoff noise

Syntax   hpsd = noisepsd(H,L)
               hpsd = noisepsd(H,L,param1,value1,param2,value2,...)
               hpsd = noisepsd(H,L,opts)
               noisepsd(H,...)

Description   hpsd = noisepsd(H,L) computes the power spectral density (PSD) at
                    the output of dfilt object or filter System object, H, occurring because of
                    roundoff noise. This noise is produced by quantization errors within the
                    filter. L is the number of trials used to compute the average. The PSD is
                    computed from the average over the L trials. The more trials you specify,
                    the better the estimate, but at the expense of longer computation time.
                    When you do not explicitly specify L, the default is 10 trials.

                    hpsd is a psd data object. To extract the PSD vector (the data from
                    the PSD) from hpsd, enter

                    get(hpsd,'data')

                    at the prompt. Plot the PSD data with plot(hpsd). The average power
                    of the output noise (the integral of the PSD) can be computed with
                    avgpower, a method of dspdata objects:

                    avgpwr = avgpower(hpsd).

                    hpsd = noisepsd(H,L,param1,value1,param2,value2,...) where H
                    can be either a dfilt object or a filter System object, specifies optional
                    parameters via propertyname/propertyvalue pairs. Valid psd object
                    property values are:

Property Name	Default Value	Description and Valid Entries
Nfft	512	Specify the number of FFT points to use to calculate the PSD.
NormalizedFrequency	true	Determine whether to use normalized frequency. Enter a logical value of `true` or `false`. Because this property is a logical value, and not a string, do not enclose the single quotation marks.
Fs	normalized	Specify the sampling frequency to use when you set `NormalizedFrequency` to `false`. Use any integer value greater than 1. Enter the value in Hz.
SpectrumType	onesided	Specify how `noisepsd` should generate the PSD. Options are `onesided` or `twosided`. If you choose a two-sided computation, you can also choose `CenterDC = true`. Otherwise, `CenterDC` must be `false`. • `onesided` converts the type to a spectrum that is calculated over half the Nyquist interval. All properties affected by the new frequency range are adjusted automatically. • `twosided` converts the type to a spectrum that is calculated over the whole Nyquist interval. All properties affected by the new frequency range are adjusted automatically.

Property Name	Default Value	Description and Valid Entries
CenterDC	false	Shift the zero-frequency component to the center of a two-sided spectrum. • When you set SpectrumType to onesided, it is changed to twosided and the data is converted to a two-sided spectrum. • Setting CenterDC to false shifts the data and the frequency values in the object so that DC is in the left edge of the spectrum. This operation does not affect the SpectrumType property setting.
Arithmetic (only for filter System objects)	ARITH	Analyze the filter System object, based on the arithmetic specified in the ARITH input. ARITH can be set to double, single, or fixed. The analysis tool assumes a double-precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state.

Note If the spectrum data you specify is calculated over half the Nyquist interval and you do not specify a corresponding frequency vector, the default frequency vector assumes that the number of points in the whole FFT was even. Also, the plot option to convert to a whole or two-sided spectrum assumes the original whole FFT length is even.

If H is a filter System object, noisepsd requires knowledge of the input data type. Analysis cannot be performed if the input data type is not available. If you do not specify the Arithmetic parameter, i.e., use the syntax [h,w] = noisepsd(H) , then the following rules apply to this method:

- The System object state is `Unlocked` — `noisepsd` performs double-precision analysis.

- The System object state is `Locked` — `noisepsd` performs analysis based on the locked input data type.

If you do specify the `Arithmetic` parameter, i.e., use the syntax `[h,w] = noisepsd(H,'Arithmetic', ARITH))`, review the following rules for this method. Which rule applies depends on the value you set for the `Arithmetic` parameter.

Value	System Object State	Rule
`ARITH = 'double'`	Unlocked	`noisepsd` performs double-precision analysis.
	Locked	`noisepsd` performs double-precision analysis.
``ARITH = `single'``	Unlocked	`noisepsd` performs single-precision analysis.
	Locked	`noisepsd` performs single-precision analysis.

Value	System Object State	Rule
ARITH = `fixed'	Unlocked	noisepsd produces an error because the fixed-point input data type is unknown.
	Locked	When the input data type is double or single, then noisepsd produces an error because since the fixed-point input data type is unknown.
		When the input data is of fixed-point type, noisepsd performs analysis based on the locked input data type.

noisepsd supports only the following filter System objects:

- dsp.FIRFilter

- dsp.IIRFilter

- dsp.AllpoleFilter

- dsp.BiquadFilter

Regardless of whether H is a dfilt object or a filter System object, hpsd = noisepsd(H,L,opts) uses an options object, opts, to specify the optional input arguments. This specification is not made using property-value pairs in the command. Use opts = noisepsdopts(H) to create the object. opts then has the noisepsd settings from H. After creating opts, you change the property values before calling noisepsd:

```
set(opts,'fs',48e3); % Set Fs to 48 kHz.
```

Regardless of whether H is a `dfilt` object or a filter System object, `noisepsd(H,...)` with no output argument launches FVTool.

Examples   Compute the PSD of the output noise caused by the quantization processes in a fixed-point, direct form FIR filter.

```
b = firgr(27,[0 .4 .6 1],[1 1 0 0]);
h = dfilt.dffir(b); % Create the filter object.
% Quantize the filter to fixed-point.
h.arithmetic = 'fixed';
hpsd = noisepsd(h);
plot(hpsd)
```

`hpsd` looks similar to the following figure—the data resulting from the noise PSD calculation. You can review the data in `hpsd.data`.
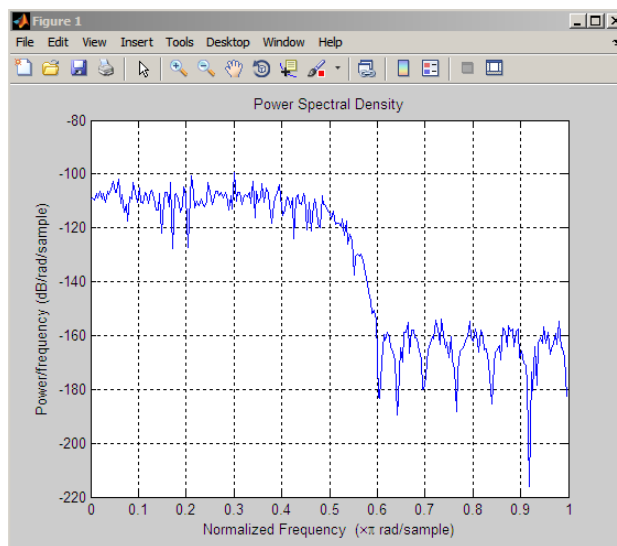
References   McClellan, et al., *Computer-Based Exercises for Signal Processing Using MATLAB 5*. Upper Saddle River, N.J.: Prentice-Hall, 1998.

See Also   filter | noisepsdopts | norm | reorder | scale | spectrum.welch

noisepsdopts

Purpose	Options for running filter output noise PSD

Syntax   `opts = noisepsdopts(H)`

Description   `opts = noisepsdopts(H)` uses the current settings in the `dfilt` object or the filter System object, `H`, to create an options object, `opts`, that contains specified options for computing the output noise PSD. You can pass `opts` to the `scale` method as an input argument to apply scaling settings to a second-order filter.

`noisepsdopts` returns the options object, `opts`, with which you can set the following properties for `noisepsd`:

Property Name	**Default Value**	**Description and Valid Entries**
`Nfft`	`512`	Specify the number of FFT points to use to calculate the PSD.
`NormalizedFrequency`	`true`	Determine whether to use normalized frequency. Enter a logical value of the logical `true` or `false`. Because this property is a logical value, and not a string, do not enclose with single quotation marks.
`Fs`	`normalized`	Specify the sampling frequency to use when you set `NormalizedFrequency` to `false`. Use any integer value greater than 1. Enter the value in Hz.

Property Name	Default Value	Description and Valid Entries
SpectrumType	onesided	Specify how noisepsd should generate the PSD. Options are onesided or twosided. If you choose a two-sided computation, you can also choose centerdc = true. Otherwise, centerdc must be false. • onesided converts the type to a spectrum that is calculated over half the Nyquist interval. All properties affected by the new frequency range are adjusted automatically. • twosided converts the type to a spectrum that is calculated over the whole Nyquist interval. All properties affected by the new frequency range are adjusted automatically.
CenterDC	false	Shift the zero-frequency component to the center of a two-sided spectrum. • When you set SpectrumType to onesided, it is changed to twosided and the data is converted to a two-sided spectrum. • Setting CenterDC to false shifts the data and the frequency values in the object so that DC is in the left edge of the spectrum.

Property Name	Default Value	Description and Valid Entries
		This operation does not effect the SpectrumType property setting.
Arithmetic (only for filter System objects)	ARITH	Analyze the filter System object, based on the arithmetic specified in the ARITH input. ARITH can be set to double, single, or fixed. The analysis tool assumes a double-precision filter when the arithmetic input is not specified and the filter System object is in an unlocked state.

See Also   noisepsd

Purpose   P-norm of filter

Syntax   l = norm(ha)
                  l = norm(ha,pnorm)
                  l = norm(hd)
                  l = norm(hd,pnorm)
                  l = norm(hm)
                  l = norm(hm,pnorm)

Description   All of the variants of norm return the filter p-norm for the object in the
                  syntax, either an adaptive filter, a digital filter, or a multirate filter.
                  When you omit the pnorm argument, norm returns the L2-norm for
                  the object.

                  Note that by Parseval's theorem, the L2-norm of a filter is equal to the
                  l2 norm. This equality is not true for the other norm variants.

                  ### For adaptfilt Objects

                  l = norm(ha) returns the L2-norm of an adaptive filter.l =
                  norm(ha,pnorm) adds the input argument pnorm to let you specify the
                  norm returned. pnorm can be either

                  • Frequency-domain norms specified by one of L1, L2, or Linf

                  • Discrete-time domain norms specified by one of l1, l2, or linf

                  ### For dfilt Objects

                  l = norm(hd) returns the L2-norm of a discrete-time filter.

                  l = norm(hd,pnorm) includes input argument pnorm that lets you
                  specify the norm returned. pnorm can be either

                  • Frequency-domain norms specified by one of L1, L2, or Linf

                  • Discrete-time domain norms specified by one of l1, l2, or linf

                  By Parseval's theorem, the L2-norm of a filter is equal to the l2 norm.
                  This equality is not true for the other norm variants.

                  IIR filters respond slightly differently to norm. When you compute the
                  l2, linf, L1, and L2 norms for an IIR filter, norm(...,L2,tol) lets you

specify the tolerance for the accuracy in the computation. For l1, l2, L2, and linf, norm uses the tolerance to truncate the infinite impulse response that it uses to calculate the norm. For L1, norm passes the tolerance to the numerical integration algorithm. Refer to Examples to see this in use. You cannot specify Linf for the norm and include the tol option.

For mfilt Objects

l = norm(hm) returns the L2-norm of a multirate filter.

l = norm(hm,pnorm) includes argument pnorm to let you specify the norm returned. pnorm can be either

- Frequency-domain norms specified by one of L1, L2, or Linf

- Discrete-time domain norms specified by one of l1, l2, or linf

Note that, by Parseval's theorem, the L2-norm of a filter is equal to the l2 norm. This equality is not true for the other norm variants.

Examples

Adaptfilt Objects

For the adaptive filter example, compute the 2-norm of an adaptfilt object, here an LMS-based adaptive filter.

```
Ha = adaptfilt.lms; % norm(ha) is zero because all coeffs are zero
% Create some data to filter to generate filter coeffs
x = randn(100,1);
d = x + randn(100,1);
[y,e] = filter(Ha,x,d);
L2 = norm(Ha); % Now norm(ha) is nonzero
```

Dfilt Objects

To demonstrate the tolerance option used with an IIR filter (dfilt object), compute the 2-norm of filter hd with a tolerance of 1e-10.

```
H =fdesign.lowpass('n,fc',5,0.4);
Hd = butter(H);
L2=norm(Hd,'l2',1e-10);
```

Mfilt Objects

In this example, compute the infinity norm of an FIR polyphase interpolator, which is an mfilt object.

```
Hm = mfilt.firinterp;
Linf = norm(Hm,'linf');
```

See Also   reorder | scale | scalecheck

normalize

Purpose	Normalize filter numerator or feed-forward coefficients
Syntax	`normalize(hq)` `g = normalize(hd)`

Description   `normalize(hq)` normalizes the filter numerator coefficients for a quantized filter to have values between -1 and 1. The coefficients of `hq` change — `normalize` does not copy `hq` and return the copy. To restore the coefficients of `hq` to the original values, use `denormalize`.

Note that for lattice filters, the feed-forward coefficients stored in the property `lattice` are normalized.

`g = normalize(hd)` normalizes the numerator coefficients for the filter `hq` to between -1 and 1 and returns the gain `g` due to the normalization operation. Calling `normalize` again does not change the coefficients. `g` always returns the gain returned by the first call to normalize the filter.

Examples   Create a direct form II quantized filter that uses second-order sections. Then use `normalize` to maximize the use of the range of representable coefficients.

```
d=fdesign.lowpass('n,fp,ap,ast',8,.5,2,40);
hd=design(d,'ellip');
hd.arithmetic ='fixed';
```

Check the filter coefficients. Note that `InitialSOSMatrix(3,2)>1`

```
InitialSOSMatrix = hd.sosMatrix;
```

Use `normalize` to modify the coefficients into the range between -1 and 1. The output `g` contains the gains applied to each section of the SOS filter.

```
g = normalize(hd);
```

None of the numerator coefficients exceed -1 or 1.

See Also   denormalize

normalizefreq

Purpose	Switch filter specification between normalized frequency and absolute frequency

Syntax	`normalizefreq(d)`
	`normalizefreq(d,flag)`
	`normalizefreq(d,`**`false`**`,fs)`

Description	`normalizefreq(d)` normalizes the frequency specifications in filter specifications object `d`. By default, the `NormalizedFrequency` property is set to `true` when you create a design object. You provide the design specifications in normalized frequency units. `normalizefreq` does not affect filters that already use normalized frequency.

If you use this syntax when `d` does not use normalized frequency specifications, all of the frequency specifications are normalized by `fs/2` so they lie between 0 and 1, where `fs` is specified in the object. Included in the normalization are the filter properties that define the filter pass and stopband edge locations by frequency:

- `F3 dB` — Used by IIR filter specifications objects to describe the passband cutoff frequency

- `Fcutoff` — Used by FIR filter specifications objects to describe the passband cutoff frequency

- `Fpass` — Describes the passband edges

- `Fstop` — Describes the stopband edges

In this syntax, `normalizefreq(d)` assumes you specified `fs` when you created `d` or changed `d` to use absolute frequency specifications.

`normalizefreq(d,flag)` where `flag` is either **true** or **false**, specifies whether the `NormalizedFrequency` property value is `true` or `false` and therefore whether the filter normalizes the sampling frequency `fs` and other related frequency specifications. `fs` defaults to 1 for this syntax.

When you do not provide the input argument `flag`, it defaults to `true`. If you set `flag` to `false`, affected frequency specifications are multiplied by `fs/2` to remove the normalization. Use this syntax to switch your

filter between using normalized frequency specifications and not using normalized frequency specifications.

normalizefreq(d,**false**,fs) lets you specify a new sampling frequency fs when you set the NormalizedFrequency property to false.

Examples   These examples demonstrate using normalizefreq in both of the major syntax applications—setting the design object frequency specifications to use absolute frequency (normalizefreq(hd,**false**,fs)) and resetting a design object to using normalized frequencies (normalizefreq(d)).

Construct a highpass filter specifications object by specifying the passband and stopband edges and the desired attenuations in the bands. By default, provide the frequency specifications in normalized values between 0 and 1.

d=fdesign.highpass(0.35, 0.45, 60, 40);

Fstop and Fpass are in normalized form, and the property NormalizedFrequency is true.

Now use normalizedfreq to convert to absolute frequency specifications, with a sampling frequency of 1000 Hz.

normalizefreq(d,false,1e3);

Both of the attenuation specifications remain the same. The passband and stopband edge definitions now appear in Hz, where the new value represents the normalized values multiplied by Fs/2, or 500 Hz.

Converting to using normalized frequencies consists of using normalizefreq with the design object d.

normalizefreq(d)

For bandstop, bandpass, and multiple band filter specifications objects, normalizefreq works the same way for all band edge definitions. When you do not provide the sampling frequency Fs as an input argument and you are converting to absolute frequency specifications, normalizefreq sets Fs to 1, as shown in this example.

normalizefreq

```
d=fdesign.bandstop(0.25,0.35,0.55,0.65,50,60);
normalizefreq(d,false)
```

See Also   fdesign.lowpass | fdesign.halfband | fdesign.highpass |
fdesign.interpolator

Purpose   Number of filter states

Syntax   n = nstates(hd)
                   n = nstates(hm)

Description   **Discrete-Time Filters**

                   n = nstates(hd) returns the number of states n in the discrete-time
                   filter hd. The number of states depends on the filter structure and the
                   coefficients.

                   Multirate Filters

                   n = nstates(hm) returns the number of states n in the multirate
                   filter hm. The number of states depends on the filter structure and the
                   coefficients.

Examples   Check the number of states for two different filters, one a direct form
                   FIR filter, the other a multirate filter.

```
h = firls(30,[0 .1 .2 .5]*2,[1 1 0 0]);
hd = dfilt.dffir(h);
NstateDF = nstates(hd);
hm = mfilt.firsrc(2,3);
NstateMR = nstates(hm);
```

See Also   mfilt

order

Purpose	Order of fixed-point filter

Syntax

```
n = order(hq)
n = order(hs)
```

Description n = order(hq) returns the order n of the quantized filter hq. When hq is a single-section filter, n is the number of delays required for a minimum realization of the filter.

When hq has more than one section, n is the number of delays required for a minimum realization of the overall filter.

n = order(hs) returns the order n of the filter System object hs. The order depends on the filter structure and the reference double-precision floating-point coefficients. hs can be one of the following filter structures:

- dsp.FIRFilter
- dsp.BiquadFilter
- dsp.FIRInterpolator
- dsp.CICInterpolator
- dsp.FIRDecimator
- dsp.CICDecimator
- dsp.FIRRateConverter

Examples Create a discrete-time filter, quantize it, and convert it to second-order section form. Then, use order to check the order of the filter.

```
[b,a] = ellip(4,3,20,.6); % Create the reference filter.
hq = dfilt.df2(b,a);
% Quantize the filter and convert to second-order sections.
set(hq,'arithmetic','fixed');

n=order(hq); % Check the order of the overall filter.
```

Purpose   Multirate parallel filter structure

Syntax   `hmp = parallel(hm1,hm2,...,hmn)`

Description `hmp = parallel(hm1,hm2,...,hmn)` returns a multirate filter `hmp` that is two or more `mfilt` objects `hm1`, `hm2`, and so on connected in a parallel structure. Each filter in the structure is one stage and all stages must have the same rate change factor.

Access the individual filters in the parallel structure by
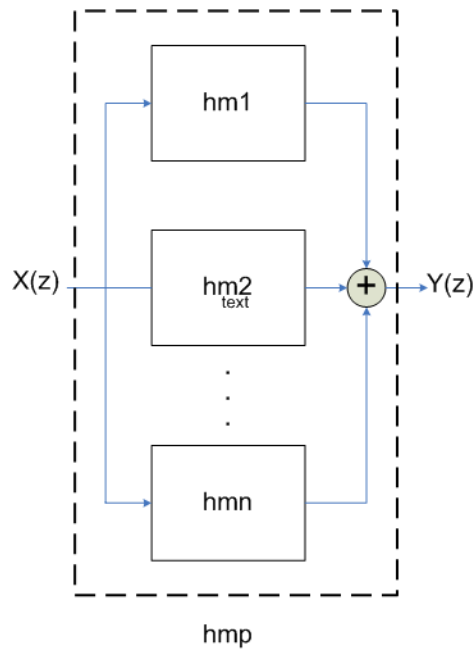
See Also   `dfilt.parallel` | `mfilt`

phasedelay

Purpose   Phase delay of filter

Syntax
```
[phi,w]=phasedelay(hfilt)
[phi,w]=phasedelay(hfilt,n)
phasedelay(hfilt)
[phi,w] = phasedelay(hs)
[phi,w] = phasedelay(hs,n)
[phi,w] = phasedelay(hs,Name,Value)
phasedelay(hs)
```

Description   phasedelay returns the phase delay based on the current filter coefficients. This section describes phasedelay operation for discrete-time filters and multirate filters. For more information about optional input arguments for phasedelay, refer to phasedelay in Signal Processing Toolbox documentation.

[phi,w]=phasedelay(hfilt) returns the phase response phi of the filter hfilt and the corresponding frequencies w at which the function evaluates the phase delay.

The phase delay is evaluated at 8192 points equally spaced around the upper half of the unit circle.

[phi,w]=phasedelay(hfilt,n) returns the phase delay phi of the filter hfilt and the corresponding frequencies w at which the function evaluates the phase delay. The phase delay is evaluated at n points equally spaced around the upper half of the unit circle.

phasedelay(hfilt) displays the phase delay of hfilt in the Filter Visualization Tool (FVTool).

[phi,w] = phasedelay(hs) returns the phase delay for the filter System object hs using 8192 samples.

[phi,w] = phasedelay(hs,n) returns the phase delay for the filter System object hs using n samples.

[phi,w] = phasedelay(hs,Name,Value) returns the phase delay with additional options specified by one or more Name,Value pair arguments.

phasedelay(hs) uses FVTool to plot the phase delay of the filter
System object hs.

**Input
Arguments**

hfilt

hfilt is either:

- A discrete-time dfilt, or multirate mfilt filter object

- A vector of discrete-time, or multirate filter objects

The multirate filter delay response is computed relative to the rate at
which the filter is running. When you specify fs (the sampling rate) as
an input argument, phasedelay assumes the filter is running at that
rate.

For multistage cascades, phasedelay forms a single-stage multirate
filter that is equivalent to the cascade. It then computes the response
relative to the rate at which the equivalent filter is running. phasedelay
does not support all multistage cascades. The function analyzes only
those cascades for which there exists an equivalent single-stage filter.

As an example, consider a 2-stage interpolator where the first stage has
an interpolation factor of 2 and the second stage has an interpolation
factor of 4. In this case, an equivalent single-stage filter exists with
an overall interpolation factor of 8. phasedelay uses this equivalent
filter for the analysis. If a sampling frequency fs is specified as an
input argument to phasedelay, the function interprets fs as the rate at
which the equivalent filter is running.

hs

Filter System object. The function supports the following filter
structures.

- dsp.FIRFilter

- dsp.BiquadFilter

- dsp.FIRInterpolator

- dsp.CICInterpolator

- `dsp.FIRDecimator`

- `dsp.CICDecimator`

- `dsp.FIRRateConverter`

n

Number of samples. For an FIR filter where n is a power of two, the computation is done faster using FFTs.

Default: 8192

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Arithmetic
`` `double' `` | `'single'` | `'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'`, the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

phasedelay

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

phi

Complex, n-element phase-delay vector. If `hfilt` is a vector of filters, `phi` is a complex, `length(hfilt)`-by-n matrix of phase-delay vectors whose columns correspond to each filter in `hfilt`. If n is not specified, the function uses a default value of 8192.

For adaptive filters, `phi` is the instantaneous phase delay.

w

Frequency vector of length n, in radians/sample. `w` consists of n points equally spaced around the upper half of the unit circle (from 0 to π radians/sample). If `hfilt` is a vector of filters, `w` is a complex, `length(hfilt)`-by-n matrix of phase-delay vectors whose columns correspond to each filter in `hfilt`. If n is not specified, the function uses a default value of 8192.

Algorithms

You can provide `fs`, the sampling frequency, as an input as well. `phasedelay` uses `fs` to calculate the delay response and plots the response to `fs/2`.

See Also

freqz | grpdelay | phasez | zerophase | zplane | freqz | fvtool | phasez | zerophase

Purpose   Unwrapped phase response for filter

Syntax   [phi,w]=phasez(hfilt)
                  [phi,w]=phasez(hfilt,n)
                  phasez(hfilt)
                  [phi,w] = phasez(hs)
                  [phi,w] = phasez(hs,n)
                  [phi,w] = phasez(hs,Name,Value)
                  phasez(hs)

Description   phasez returns the unwrapped phase response based on the current
                  filter coefficients. This section describes phasez operation for adaptive
                  filters, discrete-time filters, multirate filters, and filter System objects.
                  For more information about optional input arguments for phasez, refer
                  to phasez in Signal Processing Toolbox documentation.

                  [phi,w]=phasez(hfilt) returns the phase response phi of the filter
                  hfilt and the corresponding frequencies w at which the function
                  evaluates the phase response. The phase response is evaluated at 8192
                  points equally spaced around the upper half of the unit circle.

                  [phi,w]=phasez(hfilt,n) returns the phase response phi of the
                  filter hfilt and the corresponding frequencies w at which the function
                  evaluates the phase response. The phase response is evaluated at n
                  points equally spaced around the upper half of the unit circle.

                  phasez(hfilt) displays the phase response of hfilt in the Filter
                  Visualization Tool (FVTool).

                  [phi,w] = phasez(hs) returns a phase response for the filter System
                  object hs using 8192 samples.

                  [phi,w] = phasez(hs,n) returns a phase response for the filter
                  System object hs using n samples.

                  [phi,w] = phasez(hs,Name,Value) returns a phase response
                  with additional options specified by one or more Name,Value pair
                  arguments.

phasez

phasez(hs) uses FVTool to plot the phase response of the filter System object hs.

Input Arguments

hfilt

hfilt is either:

- An adaptive adaptfilt, discrete-time dfilt, or multirate mfilt filter object

- A vector of adaptive, discrete-time, or multirate filter objects

The multirate filter response is computed relative to the rate at which the filter is running. When you specify fs (the sampling rate) as an input argument, phasez assumes the filter is running at that rate.

For multistage cascades, phasez forms a single-stage multirate filter that is equivalent to the cascade. It then computes the response relative to the rate at which the equivalent filter is running. phasez does not support all multistage cascades. The function analyzes only those cascades for which there exists an equivalent single-stage filter.

As an example, consider a 2-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an interpolation factor of 4. In this case, an equivalent single-stage filter exists with an overall interpolation factor of 8. phasez uses this equivalent filter for the analysis. If a sampling frequency fs is specified as an input argument to phasez, the function interprets fs as the rate at which the equivalent filter is running.

hs

Filter System object. The function supports the following filter structures.

- dsp.FIRFilter

- dsp.BiquadFilter

- dsp.FIRInterpolator

- dsp.CICInterpolator

- dsp.FIRDecimator

- dsp.CICDecimator

- dsp.FIRRateConverter

n

Number of samples. For an FIR filter where n is a power of two, the computation is done faster using FFTs.

>   **Default:** 8192

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Arithmetic

`double' | 'single' | 'fixed'

For filter System object inputs only, specify the arithmetic used during analysis. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed' , the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

phi

Complex, n-element phase response vector. If `hfilt` is a vector of filters, `phi` is a complex, `length(hfilt)`-by-n matrix of phase response vectors whose columns correspond to each filter in `hfilt`. If n is not specified, the function uses a default value of 8192.

For adaptive filters, `phi` is the instantaneous phase response.

w

Frequency vector of length n, in radians/sample. `w` consists of n points equally spaced around the upper half of the unit circle (from 0 to π radians/sample). If `hfilt` is a vector of filters, `w` is a complex, `length(hfilt)`-by-n matrix of phase response vectors whose columns correspond to each filter in `hfilt`. If n is not specified, the function uses a default value of 8192.

See Also

`freqz` | `grpdelay` | `phasedelay` | `zerophase` | `zplane` | `freqz` | `fvtool` | `phasez`

polyphase

Purpose	Polyphase decomposition of multirate filter

Syntax
```
p = polyphase(hm)
polyphase(hm)
p = polyphase(hs)
p = polyphase(hs,Name,Value)
polyphase(hs)
```

Description p = polyphase(hm) returns the polyphase matrix p of the multirate filter hm.

polyphase(hm) launches the Filter Visualization Tool (FVTool) with all the polyphase subfilters to allow you to analyze each component subfilter individually.

p = polyphase(hs) returns the polyphase matrix p of the multirate filter System object hs.

p = polyphase(hs,Name,Value) returns the polyphase matrix p of the multirate filter System object hs.

polyphase(hs) launches the Filter Visualization Tool (FVTool) with all the polyphase subfilters to allow you to analyze each component subfilter individually.

Input Arguments

hm

Multirate mfilt filter object. CIC-based filters do not have coefficients, so the function does not support CIC filter structures such as mfilt.cicdecim.

hs

Filter System object. The function supports the following filter structures.

- dsp.FIRInterpolator
- dsp.CICInterpolator
- dsp.FIRDecimator

- dsp.CICDecimator

- dsp.FIRRateConverter

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Arithmetic

`double' | 'single' | 'fixed'

For filter System object inputs only, specify the arithmetic used during analysis. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed', the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the

System Object State	Coefficient Data Type	Rule
		CustomCoefficientsDataType property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

p

Polyphase matrix p of the multirate filter. Each row in the matrix represents one subfilter of the multirate filter. The first row of matrix p represents the first subfilter, the second row the second subfilter, and so on to the last subfilter.

Examples   When you create a multirate filter that uses polyphase decomposition, polyphase lets you analyze the component filters individually by returning the components as rows in a matrix.

First, create an interpolate-by-eight filter.

```
hm=mfilt.firinterp(8)

hm =

        FilterStructure: 'Direct-Form FIR Polyphase Interpolator'
              Numerator: [1x192 double]
    InterpolationFactor: 8
       PersistentMemory: false
                 States: [23x1 double]
```

In this syntax, the matrix p contains all of the subfilters for hm, one filter per matrix row.

```
p=polyphase(hm)

p =

  Columns 1 through 8

        0        0        0        0        0        0        0        0
  -0.0000   0.0002  -0.0006   0.0013  -0.0026   0.0048  -0.0081   0.0133
  -0.0001   0.0004  -0.0012   0.0026  -0.0052   0.0094  -0.0160   0.0261
  -0.0001   0.0006  -0.0017   0.0038  -0.0074   0.0132  -0.0223   0.0361
  -0.0002   0.0008  -0.0020   0.0045  -0.0086   0.0153  -0.0257   0.0415
  -0.0002   0.0008  -0.0021   0.0045  -0.0086   0.0151  -0.0252   0.0406
  -0.0002   0.0007  -0.0018   0.0038  -0.0071   0.0124  -0.0205   0.0330
  -0.0001   0.0004  -0.0011   0.0022  -0.0041   0.0072  -0.0118   0.0189


  Columns 9 through 16

        0        0        0        0   1.0000        0        0        0
  -0.0212   0.0342  -0.0594   0.1365   0.9741  -0.1048   0.0511  -0.0303
```

```
        -0.0416     0.0673    -0.1189     0.2958     0.8989    -0.1730     0.0878    -0.0527
        -0.0576     0.0938    -0.1691     0.4659     0.7814    -0.2038     0.1071    -0.0648
        -0.0661     0.1084    -0.2003     0.6326     0.6326    -0.2003     0.1084    -0.0661
        -0.0648     0.1071    -0.2038     0.7814     0.4659    -0.1691     0.0938    -0.0576
        -0.0527     0.0878    -0.1730     0.8989     0.2958    -0.1189     0.0673    -0.0416
        -0.0303     0.0511    -0.1048     0.9741     0.1365    -0.0594     0.0342    -0.0212


    Columns 17 through 24

             0          0          0          0          0          0          0          0
        0.0189    -0.0118     0.0072    -0.0041     0.0022    -0.0011     0.0004    -0.0001
        0.0330    -0.0205     0.0124    -0.0071     0.0038    -0.0018     0.0007    -0.0002
        0.0406    -0.0252     0.0151    -0.0086     0.0045    -0.0021     0.0008    -0.0002
        0.0415    -0.0257     0.0153    -0.0086     0.0045    -0.0020     0.0008    -0.0002
        0.0361    -0.0223     0.0132    -0.0074     0.0038    -0.0017     0.0006    -0.0001
        0.0261    -0.0160     0.0094    -0.0052     0.0026    -0.0012     0.0004    -0.0001
        0.0133    -0.0081     0.0048    -0.0026     0.0013    -0.0006     0.0002    -0.0000
```

Finally, using `polyphase` without an output argument opens the Filter Visualization Tool, ready for you to use the analysis capabilities of the tool to investigate the interpolator `hm`.

```
polyphase(hm)
```

In the following figure, FVTool shows the magnitude responses for the subfilters.
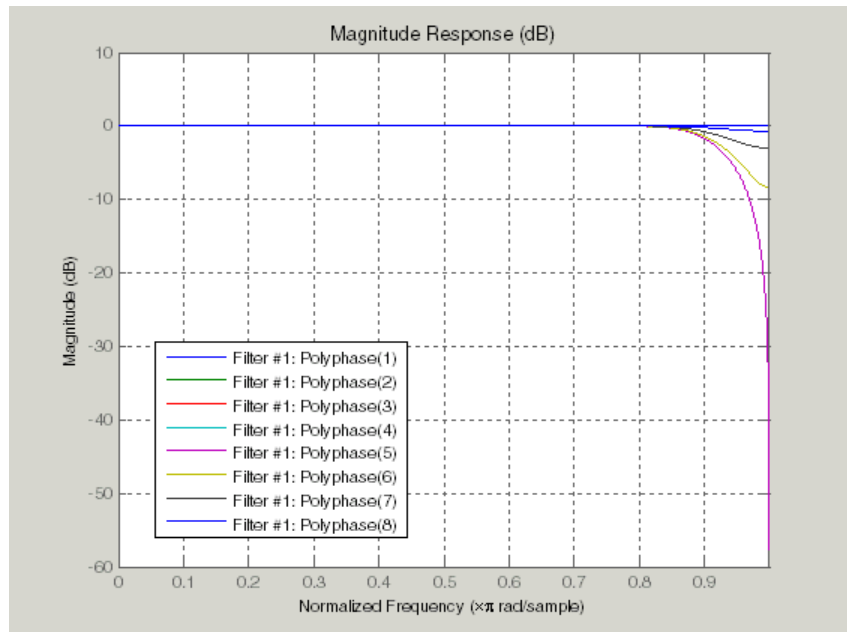
See Also   mfilt

qreport

Purpose   Most recent fixed-point filtering operation report

Syntax   rlog = qreport(h)

Description rlog = qreport(h) returns the logging report stored in the filter object
h in the object rlog. The ability to log features of the filtering operation
is integrated in the fixed-point filter object and the filter method.

Each time you filter a signal with h, new log data overwrites the results
in the filter from the previous filtering operation. To save the log from a
filtering simulation, change the name of the output argument for the
operation before subsequent filtering runs.

Note qreport requires Fixed-Point Toolbox software and that filter h is
a fixed-point filter. Data logging for fi operations is a preference you set
for each MATLAB session. To learn more about logging, LoggingMode,
and fi object preferences, refer to fipref in the Fixed-Point Toolbox
documentation.

Also, you cannot use qreport to log the filtering operations from a
fixed-point Farrow filter.

Enable logging during filtering by setting LoggingMode to on for fi
objects for your MATLAB session. Trigger logging by setting the
Arithmetic property for h to fixed, making h a fixed-point filter and
filtering an input signal.

Using Fixed-Point Filtering Logging

Filter operation logging with qreport requires some preparation in
MATLAB. Complete these steps before you use qreport.

1 Set the fixed-point object preference for LoggingMode to on for your
MATLAB session. This setting enables data logging.

```
fipref('LoggingMode','on')
```

2 Create your fixed-point filter.

3 Filter a signal with the filter.

4 Use qreport to return the filtering information stored in the filter object.

qreport provides a way to instrument your fixed-point filters and the resulting data log offers insight into how the filter responds to a particular input data signal.

Report object rlog contains a filter-structure-specific list of internal signals for the filter. Each signal contains

• Minimum and maximum values that were recorded during the last simulation. Minimum and maximum values correspond to values before quantization.

• Representable numerical range of the word length and fraction length format

• Number of overflows during filtering for that signal.

Examples   qreport depends on the LoggingMode preference for fixed-point objects. This example demonstrates the process for enabling and using qreport to log the results of filtering with a fixed-point filter. hd is a fixed-point direct-form FIR filter.

```
f = fipref('loggingmode','on');
hd = design(fdesign.lowpass,'equiripple');
hd.arithmetic = 'fixed';
fs = 1000;          % Input sampling frequency.
t = 0:1/fs:1.5;     % Signal length = 1501 samples.
x = sin(2*pi*10*t); % Amplitude = 1 sinusoid.
y = filter(hd,x);
rlog = qreport(hd)
```

View the logging report of a direct-form II, second-order sections IIR filter the same way. While this example sets loggingmode to on, you

do that only once for a MATLAB session, unless you reset the mode to `off` during the session.

```
fipref('loggingmode','on');
hd = design(fdesign.lowpass,'ellip');
hd.arithmetic = 'fixed';
y = filter(hd,rand(100,1));
rlog = qreport(hd)
```

See Also   `dfilt` | `mfilt`

Purpose   Simulink subsystem block for filter

Syntax   ```
realizemdl(hq)
realizemdl(hq,Name,Value)
```

**Description**  realizemdl(hq) generates a model of filter hq in a Simulink subsystem block using sum, gain, and delay blocks from Simulink. The properties and values of hq define the resulting subsystem block parameters.

realizemdl requires Simulink. To accurately realize models of quantized filters, use Simulink Fixed-Point.

realizemdl(hq,Name,Value) generates the model for hq with additional options specified by one or more Name,Value pair arguments. Using name-value pair arguments lets you control more fully the way the block subsystem model gets built. You can specify such details as where the block goes, what the name is, or how to optimize the block structure.

**Note** Subsystem filter blocks that you use realizemdl to create support sample-based input and output only. You cannot input or output frame-based signals with the block.

**Input Arguments**

**hq**

One of the following types of filter object or filter System object:

- Discrete-time filter (dfilt) object

- Multirate filter (mfilt) object.

- Filter System object. The function supports the following structures:

  - dsp.FIRFilter

  - dsp.BiquadFilter

  - dsp.FIRInterpolator

- dsp.CICInterpolator

- dsp.FIRDecimator

- dsp.CICDecimator

- dsp.FIRRateConverter

## Name-Value Pair Arguments

### Destination

'current' (default) | 'new' | subsystemname

Specify whether to add the block to your current Simulink model or create a new model to contain the block. If you provide the name of a current subsystem in *subsystemname*, realizemdl adds the new block to the specified subsystem.

### Blockname

'filter' (default)

Provide the name for the new subsystem block. By default the block is named Filter.

### MapCoeffstoPorts

'off' (default) | 'on'

Specify whether to map the coefficients of the filter to the ports of the block.

### MapStates

'off' (default) | 'on'

Specify whether to apply the current filter states to the realized model. Such specification allows you to save states from a filter object you may have used or configured in a specific way. The default setting of 'off' means the states are not transferred to the model. Setting the property to 'on' preserves the current filter states in the realized model.

### OverwriteBlock

'off' (default) | 'on'

Specify whether to overwrite an existing block with the same name or create a new block.

### OptimizeZeros
`'off'` (default) | `'on'`

Specify whether to remove zero-gain blocks.

### OptimizeOnes
`'off'` (default) | `'on'`

Specify whether to replace unity-gain blocks with direct connections.

### OptimizeNegOnes
`'off'` (default) | `'on'`

Specify whether to replace negative unity-gain blocks with a sign change at the nearest sum block.

### OptimizeDelayChains
`'off'` (default) | `'on'`

Specify whether to replace delay chains made up of $n$ unit delays with a single delay by $n$.

### CoeffNames
`{'Num'}` (default FIR) | `{'Num','Den'}` (default direct form IIR) | `{'Num','Den','g'}` (default IIR SOS) | `{'Num_1','Num_2','Num_3'...}` (default multistage) | `{'K'}` (default form lattice)

Specify the coefficient variable names as string variables in a cell array. `MapCoeffsToPorts` must be set to `'on'` for this property to apply.

### InputProcessing
`'columns as channels'` (default) | `'elements as channels'` | `'inherited'`

Specify sample-based (`'elements as channels'`) or frame-based (`'columns as channels'`) processing.

# realizemdl

---

**Note** The `inherited` option will be removed in a future release.

---

**RateOption**

`'enforce single rate'` (default) | `'allow multirate'`

Specify how the block adjusts the rate at the output to accommodate the reduced number of samples. This parameter applies only when `InputProcessing` is `'columns as channels'`.

**Arithmetic**

`'double' | 'single' | 'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'` , the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

**Details for Fixed-Point Arithmetic**

| System Object State | Coefficient Data Type | Rule |
|---|---|---|
| Unlocked | `'Same as input'` | The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Unlocked | `'Custom'` | The function performs fixed-point analysis based on the setting of the |

| System Object State | Coefficient Data Type | Rule |
|---|---|---|
| | | CustomCoefficientsDataType property. |
| Locked | `'Same as input'` | When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption. |
| Locked | `'Custom'` | The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property. |

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

**Examples**   Realize Simulink model of lowpass Butterworth filter:

```
d = fdesign.lowpass('N,F3dB',4,0.25);
Hd = design(d,'butter');
realizemdl(Hd);
```

The realized model is shown in the figure:

Realize Simulink model with coefficients mapped to ports:

```
d = fdesign.lowpass('N,F3dB',4,0.25);
Hd = design(d,'butter');
%Realize Simulink model and export coefficients
realizemdl(Hd,'MapCoeffsToPorts','on');
```

In this case, the filter is an IIR filter with a direct form II second-order sections structure. Setting MapCoeffstoPorts to 'on' exports the numerator coefficients, the denominator coefficients, and the gains to the MATLAB workspace using the default variable names Num, Den, and g. Each column of Num and Den represents one second-order section. You can modify the filter coefficients directly in the MATLAB workspace providing tunability to the realized Simulink model.

See Also   design | fdesign

Purpose	Number of samples of delay introduced by buffering and unbuffering operations

Syntax

```
d = rebuffer_delay(f,n,v)
d = rebuffer_delay(f,n,v,'mode')
```

Description $d = rebuffer\_delay(f,n,v)$ returns the delay, in samples, introduced by the Buffer or Unbuffer block in multitasking operations.

$d = rebuffer\_delay(f,n,v,'mode')$ returns the delay, in samples, introduced by the Buffer or Unbuffer block in the specified tasking mode.

Input Arguments

f

Frame size of the input to the Buffer or Unbuffer block.

n

Size of the output buffer. Specify one of the following:

- The value of the **Output buffer size** parameter, if you are computing the delay introduced by a Buffer block.

- 1, if you are computing the delay introduced by an Unbuffer block.

v

Amount of buffer overlap. Specify one of the following:

- The value of the **Buffer overlap** parameter, if you are computing the delay introduced by a Buffer block.

- 0, if you are computing the delay introduced by an Unbuffer block.

'mode'

The tasking mode of the model. Specify one of the following options:

- 'singletasking'

- 'multitasking'

rebuffer_delay

Default: `'multitasking'`

Definitions

Multitasking

When you run a model in `MultiTasking` mode, Simulink processes groups of blocks with the same execution priority through each stage of simulation based on task priority. Multitasking mode helps to create valid models of real-world multitasking systems, where sections of your model represent concurrent tasks. The **Tasking mode for periodic sample times** parameter on the Solver pane of the Configuration Parameters dialog box controls this setting.

Singletasking

When you run a model in `SingleTasking` mode, Simulink processes all blocks through each stage of simulation together. The **Tasking mode for periodic sample times** parameter on the Solver pane of the Configuration Parameters dialog box controls this setting.

Examples

Compute the delay introduced by a Buffer block in a multitasking model:

1 Open a model containing a Buffer block. For this example, open the ex_buffer_tut4 model by typing ex_buffer_tut4 at the MATLAB command line.

2 Double-click the Buffer block to open the block mask. Verify that you have the following settings:

- **Output buffer size** = 3
- **Buffer overlap** = 1
- **Initial conditions** = 0

Based on these settings, two of the required inputs to the rebuffer_delay function are as follows:

- n = 3
- v = 1

3 To determine the frame size of the input signal to the Buffer block, open the Signal From Workspace block mask. Verify that you have the following settings:

- **Signal** = sp_examples_src

- **Sample time** = 1

- **Samples per frame** = 4

Because **Samples per frame** = 4, you know the f input to the rebuffer_delay function is 4.

4 After you verify the values of all the inputs to the rebuffer_delay function, determine the delay that the Buffer block introduces in this multitasking model. To do so, type the following at the MATLAB command line:

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

d =
    8
```

Compute the delay introduced by an Unbuffer block in a multitasking model:

1 Open a model containing an Unbuffer block. For this example, open the ex_unbuffer_ref1 model by typing ex_unbuffer_ref1 at the MATLAB command line.

2 To determine the frame size of the input to the Buffer block, open the Signal From Workspace block mask by double-clicking the block in your model. Verify that you have the following settings:

- **Signal** = sp_examples_src

- **Sample time** = 1

- **Samples per frame** = 3

Because **Samples per frame** = 3, you know the f input to the rebuffer_delay function is 3.

3 Use the rebuffer_delay function to determine the amount of delay that the Unbuffer block introduces in this multitasking model. To compute the delay introduced by the Unbuffer block, use f = 3, n = 1 and v = 0.

```
d = rebuffer_delay(3,1,0)

d =
    3
```

See Also   Buffer | Unbuffer

How To   • "Buffer Delay and Initial Conditions"

Purpose	Reference filter for fixed-point or single-precision filter
Syntax	`href = reffilter(hd)`
Description	`href = reffilter(hd)` returns a new filter `href` that has the same structure as `hd`, but uses the reference coefficients and has its arithmetic property set to `double`. Note that `hd` can be either a fixed-point filter (arithmetic property set to 'fixed', or a single-precision floating-point filter whose arithmetic property is 'single').

`reffilter(hd)` differs from `double(hd)` in that

- the filter `href` returned by `reffilter` has the reference coefficients of `hd`.

- `double(hd)` returns the quantized coefficients of `hd` represented in double-precision.

To check the performance of your fixed-point filter, use `href = reffilter(hd)` to quickly have the floating-point, double-precision version of `hd` available for comparison.

Examples   Compare several fixed-point quantizations of a filter with the same double-precision floating-point version of the filter.

```
h = dfilt.dffir(firceqrip(87,.5,[1e-3,1e-6])); % Lowpass filter.
h1 = copy(h); h2 = copy(h); % Create copies of h.
h.arithmetic = 'fixed';    % Set h to filter using fixed-point...
                           % arithmetic.
h1.arithmetic = 'fixed'; % Same for h1.
h2.arithmetic = 'fixed'; % Same for h2.
h.CoeffWordLength  = 16; % Use 16 bits to represent the...
                           % coefficients.
h1.CoeffWordLength = 12; % Use 12 bits to represent the...
                           % coefficients.
h2.CoeffWordLength =  8; % Use 8 bits to represent the...
                           % coefficients.
href = reffilter(h);
hfvt = fvtool(href,h,h1,h2);
```

```
set(hfvt,'ShowReference','off'); % Reference displayed once
                                 % already.
legend(hfvt,'Reference filter','16-bits','12-bits','8-bits');
```

The following plot, taken from FVTool, shows `href`, the reference filter, and the effects of using three different word lengths to represent the coefficients.

As expected, the fidelity of the fixed-point filters suffers as you change the representation of the coefficients. With `href` available, it is easy to see just how the fixed-point filter compares to the ideal.

See Also   double

Purpose   Rearrange sections in SOS filter

Syntax
```
reorder(hd,order)
reorder(hd,numorder,denorder)
reorder(hd,numorder,denorder,svorder)
reorder(hd,filter_type)
reorder(hd,dir_flag)
reorder(hd,dir_flag,sv)
reorder(hs,...)
reorder(hs,...,Name,Value)
```

Description   reorder(hd,order) rearranges the sections of filter hd using the vector of indices provided in order.

reorder(hd,numorder,denorder) reorders the numerator and denominator separately using the vectors of indices in numorder and denorder.

reorder(hd,numorder,denorder,svorder) specifies that the scale values can be independently reordered.

reorder(hd,filter_type) reorders hd in a way suitable for the specified filter type. This reordering mode can be especially helpful for fixed-point implementations where the order of the filter sections can significantly affect your filter performance.

reorder(hd,dir_flag) specifies rearranges the sections according to proximity to the origin of the poles of the sections.

reorder(hd,dir_flag,sv) reorders scale values in addition to rearranging sections according to pole-origin proximity.

reorder(hs,...) rearranges the sections of the filter System object hs according to any of the preceding input arguments.

reorder(hs,...,Name,Value) rearranges the sections of the filter System object hs with additional options specified by one or more Name,Value pair arguments.

reorder

Input Arguments

hd

Discrete-time `dfilt.df1sos`, `dfilt.df2tsos`, `dfilt.df2sos`, or `dfilt.df1tsos`filter object.

hs

`dsp.BiquadFilter` filter System object.

order

Vector of indices used to reorder the filter sections. `order` does not need to contain all of the indices of the filter. Omitting one or more filter section indices removes the omitted sections from the filter. You can use a logical array to remove sections from the filter, but not to reorder it.

numorder

Vector of indices used to reorder the numerator. `numorder` and `denorder` must be the same length.

denorder

Vector of indices used to reorder the numerator. `numorder` and `denorder` must be the same length.

svorder

Independent reordering of scale values. When `svorder` is not specified, the scale values are reordered with the numerator. The output scale value always remains on the end when you use the argument `numorder` to reorder the scale values.

filter_type
`'auto'` | `'bandpass'` | `'bandstop'` | `'highpass'` | `'lowpass'`

Filter type. The `'auto'` option and automatic ordering only apply to filters that you used `fdesign` to create. With the `'auto'` option as an input argument, `reorder` automatically rearranges the filter sections depending on the specification response type of the design.

dir_flag

`'down' | 'up'`

Pole direction flag. When `dir_flag` is `'up'`, the first filter section contains the poles closest to the origin, and the last section contains the poles closest to the unit circle. When `ir_flag` is `'down'`, the sections are ordered in the opposite direction. `reorder` always pairs zeros with the poles closest to them.

sv

`'poles' | 'zeros'`

Reorder scale values according to poles or zeros. By default the scale values are not reordered when you use the `dir_flag` input argument.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Arithmetic

`` `double' | 'single' | 'fixed' ``

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'`, the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

Examples
Being able to rearrange the order of the sections in a filter can be a powerful tool for controlling the filter process. This example uses reorder to change the sections of a df2sos filter. Let reorder do the reordering automatically in the first example. In the second, use reorder to specify the new order for the sections.

First use the automatic reordering option on a lowpass filter.

```
d = fdesign.lowpass('n,f3db',15,0.75);
hd = design(d,'butter');
hdreorder=reorder(hd,'auto');
```

For another example of using reorder, create an SOS filter in the direct form II implementation.

```
[z,p,k] = butter(15,.5);
[sos, g] = zp2sos(z,p,k);
hd = dfilt.df2sos(sos,g);
reorder(hd, [1 3:7 2 8]);
hfvt = fvtool(hd, 'analysis', 'coefficients');
```

Reorder the sections by moving the second section to be between the seventh and eighth sections.

```
reorder(hd, [1 3:7 2 8]);
hfvt = fvtool(hd, 'analysis', 'coefficients');
```

Remove the third, fourth, and seventh sections.

```
hd1 = copy(hd);
reorder(hd1, logical([1 1 0 0 1 1 0 1]));
setfilter(hfvt, hd1);
```

Move the first filter to the end, and remove the eighth section

```
    hd2 = copy(hd);
reorder(hd2, [2:7 1]);
setfilter(hfvt, hd2);
```

Move the numerator and denominator independently.

```
hd3 = copy(hd);
reorder(hd3, [1 3:8 2], [1:8]);
setfilter(hfvt, hd3);
```

References Schlichthärle, Dietrich, *Digital Filters Basics and Design*,
Springer-Verlag Berlin Heidelberg, 2000.

See Also cumsec | scale | scaleopts

Purpose   Reset filter properties to initial conditions

Syntax
```
reset(ha)
reset(hd)
reset(hm)
```

Description reset(ha) resets all the properties of the adaptive filter ha that are
updated when filtering to the value specified at construction. If you
do not specify a value for any particular property when you construct
an adaptive filter, the property value for that property is reset to the
default value for the property.

reset(hd) resets all the properties of the discrete-time filter hd to their
factory values that are modified when you run the filter. In particular,
the States property is reset to zero.

reset(hm) resets all the properties of the multirate filter hm to their
factory value that are modified when the filter is run. In particular, the
States property is reset to zero when hm is a decimator. Additionally,
the filter internal properties are also reset to their factory values.

Examples   Denoise a sinusoid and reset the filter after filtering with it.

```
h = adaptfilt.lms(5,.05,1,[0.5,0.5,0.5,0.5,0.5]);
n = filter(1,[1 1/2 1/3],.2*randn(1,2000));
d = sin((0:1999)*2*pi*0.005) + n; % Noisy sinusoid
x = n;
[y,e]= filter(h,x,d);              % e has denoised signal
disp(h)
reset(h); % Reset the coefficients and states.
disp(h)
```

See Also   set

scale

Purpose   Scale sections of SOS filter

Syntax
```
scale(hd)
scale(hd,pnorm)
scale(hd,pnorm,Name,Value)
scale(hd,pnorm,opts)
scale(hs)
```

Description   `scale(hd)` scales the second-order section filter `hd` using peak magnitude response scaling (L-infinity, `'Linf'`). This scaling reduces the possibility of overflows when your filter `hd` operates in fixed-point arithmetic mode.

`scale(hd,pnorm)` specifies the norm used to scale the filter.

`scale(hd,pnorm,Name,Value)` scales the SOS filter with additional options specified by one or more `Name,Value` pair arguments.

`scale(hd,pnorm,opts)` uses an input scale options object `opts` to specify the optional scaling parameters instead of specifying parameter-value pairs.

`scale(hs)` scales the filter System object `hs`. You can also use `pnorm`, name-value pair arguments, or scale option objects with this syntax.

Input Arguments

hd

Discrete-time `dfilt` filter object.

With the `Arithmetic` property of `hd` set to `double` or `single`, the filter uses the default values for all options that you do not specify explicitly. When you set `Arithmetic` to `fixed`, the values used for the scaling options are set according to the settings in filter `hd`. However, if you specify a scaling option different from the settings in `hd`, the filter uses your explicit option selection for scaling purposes, but does not change the property setting in `hd`.

hs

`dsp.BiquadFilter` filter System object.

pnorm

`'l1'` | `'l2'` | `'linf'` | `'L1'` | `'L2'` | `'Linf'` (default)

Discrete-time-domain norm or a frequency-domain norm.

Valid time-domain norm values for `pnorm` are `'l1'`, `'l2'`, and `'linf'`. Valid frequency-domain norm values are `'L1'`, `'L2'`, and `'Linf'`. The `'L2'` norm is equal to the `'l2'` norm (by Parseval's theorem), but this equivalency does not hold for other norms — `'l1'` is not the same as `'L1'` and `'Linf'` is not the same as `'linf'`.

Filter norms can be ordered in terms of how stringent they are, as follows from most stringent to least: `'l1'`, `'Linf'`, `'l2'` (`'L2'`), `'linf'`. Using `'l1'`, the most stringent scaling, produces a filter that is least likely to overflow, but has the worst signal-to-noise ratio performance. The default scaling `'Linf'` is the most commonly used scaling norm.

opts

Scale options object. You can create the `opts` object using the `scaleopts` function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Arithmetic

`` `double' `` | `'single'` | `'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'`, the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

MaxNumerator

Maximum allowed value for numerator coefficients.

Default: 2

MaxScaleValue

Maximum allowed scale values. The filter applies the MaxScaleValue limit only when you set ScaleValueConstraint to a value other than unit (the default setting). Setting MaxScaleValue to any numerical value automatically changes the ScaleValueConstraint setting to none.

Default: 'Not Used'

NumeratorConstraint
'none' (default) | 'normalize' | 'po2' | 'unit'

Specifies whether and how to constrain numerator coefficient values.

OverflowMode
satall | saturate | wrap (default)

Sets the way the filter handles arithmetic overflow situations during scaling. If your device does not have guard bits available, and you are using saturation arithmetic for filtering, use 'satall' instead of 'saturate'.

ScaleValueConstraint
'none' | 'po2' | 'unit' (default)

Specify whether to constrain the filter scale values, and how to constrain them. Choosing 'unit' for the constraint disables the MaxScaleValue

scale

property setting. `'po2'` constrains the scale values to be powers of 2, while `'none'` removes any constraint on the scale values.

sosReorder

auto (default) | down | none | up

Reorder filter sections prior to applying scaling.

Examples

Demonstrate the Linf-norm scaling of a lowpass elliptic filter with second-order sections. Start by creating a lowpass elliptical filter in zero, pole, gain (z,p,k) form.

```
[z,p,k] = ellip(5,1,50,.3);
[sos,g] = zp2sos(z,p,k);
hd = dfilt.df2sos(sos,g);
scale(hd,'linf','scalevalueconstraint','none','maxscalevalue',2)
```

References

Dehner, G.F. "Noise Optimized Digital Filter Design: Tutorial and Some New Aspects." *Signal Processing*. Vol. 83, Number 8, 2003, pp. 1565–1582.

See Also

cumsec | norm | reorder | scalecheck | scaleopts

Purpose   Check scaling of SOS filter

Syntax   s = scalecheck(hd,pnorm)
                 s = scalecheck(hs,pnorm)
                 s = scalecheck(hs,pnorm,Name,Value)

Description s = scalecheck(hd,pnorm) checks the scaling second-order section
                 (SOS) dfilt filter object hd.

                 s = scalecheck(hs,pnorm) checks the scaling of the filter System
                 object hs.

                 s = scalecheck(hs,pnorm,Name,Value) checks the scaling of the
                 filter System object hs with additional options specified by one or more
                 Name,Value pair arguments.

Input   **hd**
Arguments
                 Discrete-time dfilt.df1sos, dfilt.df2tsos, dfilt.df2sos, or
                 dfilt.df1tsosfilter object.

                 hs

                 dsp.BiquadFilter filter System object.

                 pnorm
                 'l1' | 'l2' | 'linf' | 'L1' | 'L2' | 'Linf'

                 Discrete-time-domain norm or a frequency-domain norm.

                 Valid time-domain norm values for pnorm are 'l1', 'l2', and 'linf'.
                 Valid frequency-domain norm values are 'L1', 'L2', and 'Linf'. The
                 'L2' norm is equal to the 'l2' norm (by Parseval's theorem), but this
                 equivalency does not hold for other norms — 'l1' is not the same as
                 'L1' and 'Linf' is not the same as 'linf'.

                 Name-Value Pair Arguments

                 Specify optional comma-separated pairs of Name,Value arguments,
                 where Name is the argument name and Value is the corresponding

value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Arithmetic

`'double' | 'single' | 'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'`, the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

System Object State	Coefficient Data Type	Rule
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

Output Arguments

s

Filter scaling for a given p-norm. An optimally scaled filter has partial norms equal to one. In such cases, s contains all ones.

For direct-form I (dfilt.df1sos) and direct-form II transposed (dfilt.df2tsos) filters, the function returns the p-norm of the filter computed from the filter input to the output of each second-order section. Therefore, the number of elements in s is one less than the number of sections in the filter. This p-norm computation does not include the trailing scale value of the filter, which you can find by entering hd.scalevalue(end) at the MATLAB prompt.

scalecheck

For direct-form II (`dfilt.df2sos`) and direct-form I transposed (`dfilt.df1tsos`) filters, the function returns a row vector whose elements contain the p-norm from the filter input to the input of the recursive part of each second-order section. This computation of the p-norm corresponds to the input to the multipliers in these filter structures. These inputs correspond to the locations in the signal flow where overflow should be avoided.

When `hd` has nontrivial scale values, that is, if any scale values are not equal to one, `s` is a two-row matrix, rather than a vector. The first row elements of `s` report the p-norm of the filter computed from the filter input to the output of each second-order section. The elements of the second row of `s` contain the p-norm computed from the input of the filter to the input of each scale value between the sections. For `df2sos` and `df1tsos` filter structures, the last numerator and the trailing scale value for the filter are not included when `scalecheck` checks the scaling.

Examples   Check the Linf-norm scaling of a filter.

```
% Create filter design specifications
hs = fdesign.lowpass;
object.
hd = ellip(hs);       % Design an elliptic sos filter
scale(hd,'Linf');
s = scalecheck(hd,'Linf')
```

In another form:

```
[b,a]=ellip(10,.5,20,0.5);
[s,g]=tf2sos(b,a);
hd=dfilt.df1sos(s,g)


hd =


        FilterStructure: 'Direct-Form I, Second-Order Sections'
              Arithmetic: 'double'
                sosMatrix: [5x6 double]
              ScaleValues: [6x1 double]
```

```
          PersistentMemory: false
                    States: [1x1 filtstates.dfiir]

1x1 struct array with no fields.

scalecheck(hd,'Linf')

ans =

    0.7631    0.9627    0.9952    0.9994    1.0000
```

See Also   norm | reorder | scale | scaleopts

scaleopts

Purpose	Options for scaling SOS filter

Syntax	`opts = scaleopts(hd)`
	`opts = scaleopts(hs)`
	`opts = scaleopts(hs,Name,Value)`

Description	`opts = scaleopts(hd)` uses the current settings in the `dfilt` filter `hd` to create an options object `opts` that contains specified scaling options for second-order section scaling. You can pass `opts` as an input to `scale` to apply scaling settings to a second-order filter. `opts = scaleopts(hs)` creates an options object based on the current settings of the filter System object `hs`. `opts = scaleopts(hs,Name,Value)` creates an options object with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

hd

Discrete-time `dfilt.df1sos`, `dfilt.df2tsos`, `dfilt.df2sos`, or `dfilt.df1tsos`filter object.

hs

`dsp.BiquadFilter` filter System object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Arithmetic
`` `double' `` | `'single'` | `'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function

performs double- or single-precision analysis. When you specify
`'fixed'` , the arithmetic changes depending on the setting of the
`CoefficientDataType` property and whether the System object is
locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis

System Object State	Coefficient Data Type	Rule
		based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type.

Output Arguments

opts

Scaling object. The following table lists the properties of opts.

Parameter	Default	Description and Valid Value
MaxNumerator	2	Maximum allowed value for numerator coefficients.
MaxScaleValue	No default value	Maximum allowed scale values. The filter applies the MaxScaleValue limit only when you set ScaleValueConstraint to a value other than unit. Setting MaxScaleValue to a numerical value automatically changes the ScaleValueConstraint setting to none.

Parameter	Default	Description and Valid Value
NumeratorConstraint	none	Specifies whether and how to constrain numerator coefficient values. Options are none, normalize, po2, and unit.
OverflowMode	wrap	Sets the way the filter handles arithmetic overflow situations during scaling. Choose one ofwrap or saturate or satall.
ScaleValueConstraint	unit	Specify whether to constrain the filter scale values, and how to constrain them. Valid options are none, po2, and unit.

Examples   From a filter hd, you can create an options scaling object that contains the scaling options settings you require.

```
[b,a]=ellip(10,.5,20,0.5);
[s,g]=tf2sos(b,a);
hd=dfilt.df1sos(s,g)
opts=scaleopts(hd)

opts =

          MaxNumerator: 2
    NumeratorConstraint: 'none'
          OverflowMode: 'wrap'
    ScaleValueConstraint: 'unit'
```

```
MaxScaleValue: 'Not used'
```

See Also   cumsec | norm | reorder | scale | scalecheck

Purpose   Configure filter for integer filtering

Syntax
```
set2int(h)
set2int(h,coeffwl)
set2int(...,inwl)
g = set2int(...)
```

Description   These sections apply to both discrete-time (`dfilt`) and multirate (`mfilt`) filters.

set2int(h) scales the filter coefficients to integer values and sets the filter coefficient and input fraction lengths to zero.

set2int(h,coeffwl) uses the number of bits specified by coeffwl as the word length it uses to represent the filter coefficients.

set2int(...,inwl) uses the number of bits specified by coeffwl as the word length it uses to represent the filter coefficients and the number of bits specified by inwl as the word length to represent the input data.

g = set2int(...) returns the gain g introduced into the filter by scaling the filter coefficients to integers. g is always calculated to be a power of 2.

Note set2int does not work with CIC decimators or interpolators because they do not have coefficients.

Examples   These examples demonstrate some uses and ideas behind set2int.

The second parts of both examples depend on the following — after you filter a set of data, the input data and output data cover the same range of values, unless the filter process introduces gain in the output. Converting your filter object to integer form, and then filtering a set of data, does introduce gain into the system. When the examples refer to resetting the output to the same range as the input, the examples are accounting for this added gain feature.

Discrete-Time Filter Example

Two parts comprise this example. Part 1 compares the step response of an FIR filter in both the fractional and integer filter modes. Fractional mode filtering is essentially the opposite of integer mode. Integer mode uses a filter which has coefficients represented by integers. Fractional mode filters have coefficients represented in fractional form (nonzero fraction length).

```
b = firrcos(100,.25,.25,2,'rolloff','sqrt');
hd = dfilt.dffir(b);
hd.Arithmetic = 'fixed';
hd.InputFracLength = 0; % Integer inputs.
x = ones(100,1);
yfrac = filter(hd,x); % Fractional mode output.
g = set2int(hd);      % Convert to integer coefficients.
yint = filter(hd,x);  % Integer mode output.
```

Note that `yint` and `yfrac` are `fi` objects. Later in this example, you use the `fi` object properties `WordLength` and `FractionLength` to work with the output data.

Now use the gain `g` to rescale the output from the integer mode filter operation. Verify that the scaled integer output is equal to the fractional output.

```
yints = double(yint)/g;
MaxDiff=max(abs(yints-double(yfrac)));
```

Verify that the scaled integer output is equal to the fractional output.

```
 max(abs(yints-double(yfrac)))
```

In part two, the example reinterprets the output binary data, putting the input and the output on the same scale by weighting the most significant bits in the input and output data equally.

```
WL = yint.WordLength;
FL = yint.Fractionlength + log2(g);
```

```
yints2 = fi(zeros(size(yint)),true,WL,FL);
yints2.bin = yint.bin;
MaxDiff=max(abs(double(yints2)-double(yfrac)));
```

Multirate Filter Example

This two-part example starts by comparing the step response of a multirate filter in both fractional and integer modes. Fractional mode filtering is essentially the opposite of integer mode. Integer mode uses a filter which has coefficients represented by integers. Fractional mode filters have coefficients in fractional form with nonzero fraction lengths.

```
hm = mfilt.firinterp;
hm.Arithmetic = 'fixed';
hm.InputFracLength = 0; % Integer inputs.
x = ones(100,1);
yfrac = filter(hm,x);  % Fractional mode output.
g = set2int(hm);        %Convert to integer coefficients.
yint = filter(hm,x);   % Integer mode output.
```

Note that `yint` and `yfrac` are `fi` objects. In part 2 of this example, you use the `fi` object properties `WordLength` and `FractionLength` to work with the output data.

Now use the gain `g` to rescale the output from the integer mode filter operation.

```
yints = double(yint)/g;
```

Verify that the scaled integer output is equal to the fractional output.

```
max(abs(yints-double(yfrac)))
```

Part 2 demonstrates reinterpreting the output binary data by using the properties of `yint` to create a scaled version of `yint` named `yints2`. This process puts `yint` and `yints2` on the same scale by weighing the most significant bits of each object equally.

```
wl = yint.wordlength;
fl = yint.fractionlength + log2(g);
```

```
yints2 = fi(zeros(size(yint)),true,wl,fl);
yints2.bin = yint.bin;
max(abs(double(yints2)-double(yfrac)))
```

See Also   mfilt

Purpose   Specifications for filter specification object

Syntax   setspecs(D,specvalue1,specvalue2,...)
                 setspecs(D,Specification,specvalue1,specvalue2,...)
                 setspecs(...Fs)
                 setspecs(...,MAGUNITS)

Description setspecs(D,specvalue1,specvalue2,...) sets the specifications
                in filter specification object, D, in the same order they appear in the
                Specification property.

                setspecs(D,Specification,specvalue1,specvalue2,...) changes
                the specifications for an existing filter specification object and sets
                values for the new Specification property.

                setspecs(...Fs) specifies the sampling frequency, Fs, in Hz. The
                sampling frequency must be a scalar trailing all other specifications.
                Entering a sampling frequency causes all other frequency specifications
                to be in Hz.

                setspecs(...,MAGUNITS) specifies the units for any magnitude
                specifications. MAGUNITS can be one of the following: 'linear', 'dB',
                or 'squared'. The default is 'dB'. The magnitude specifications are
                always converted and stored in dB regardless of how the units are
                specified.

                Use SET(D,'SPECIFICATION') to get the list of all available
                specification types for the filter specification object, D.

Examples   Construct a lowpass filter with specifications for the filter order and
                cutoff frequency (-6 dB). Use setspecs after construction to set the
                values of the filter order and cutoff frequency. Display the values in
                the MATLAB command window.

                ```
                D = fdesign.lowpass('N,Fc');
                setspecs(D,10,0.2);
                D.FilterOrder
                D.Fcutoff
                ```

Construct a highpass filter with specifications for the numerator order, denominator order, and 3-dB frequency. Assume the sampling frequency is 1 kHz. Use `setspecs` to set the numerator and denominator orders to 6. Set the 3-dB frequency to 250 Hz. In order to use frequency specifications in Hz, specify the sampling frequency as a trailing scalar.

```
D = fdesign.highpass('Nb,Na,F3dB');
setspecs(D,6,6,250,1000);
```

See Also   design | designmethods | designopts | fdesign

Purpose	Convert quantized filter to second–order sections (SOS) form

Syntax

```
Hq2 = sos(Hq)
Hq2 = sos(Hq, order)
Hq2 = sos(Hq, order, scale)
```

Description   Hq2 = sos(Hq) returns a quantized filter Hq2 that has second-order sections and the dft2 structure. Use the same optional arguments used in tf2sos.

Hq2 = sos(Hq, order) specifies the order of the sections in Hq2, where order is either of the following strings:

- 'down' — to order the sections so the first section of Hq2 contains the poles closest to the unit circle (L_∞ norm scaling)

- 'up' — to order the sections so the first section of Hq2 contains the poles farthest from the unit circle (L_2 norm scaling and the default)

Hq2 = sos(Hq, order, scale) also specifies the desired scaling of the gain and numerator coefficients of all second-order sections, where scale is one of the following strings:

- 'none' — to apply no scaling (default)

- 'inf' — to apply infinity-norm scaling

- 'two' — to apply 2-norm scaling

Use infinity-norm scaling in conjunction with up-ordering to minimize the probability of overflow in the filter realization. Consider using 2-norm scaling in conjunction with down-ordering to minimize the peak round-off noise.

When Hq is a fixed-point filter, the filter coefficients are normalized so that the magnitude of the maximum coefficient in each section is 1. The gain of the filter is applied to the first scale value of Hq2.

sos uses the direct form II transposed (dft2) structure to implement second- order section filters.

Examples
```
[b,a]=butter(8,.5);
Hq = dfilt.df2t(b,a);
Hq.arithmetic = 'fixed';
Hq1 = sos(Hq)

Hq1 =

            FilterStructure: 'Direct-Form II Transposed, Second-Order Sections'
                 Arithmetic: 'double'
                  sosMatrix: [4x6 double]
                ScaleValues: [0.00927734375;1;1;1;1]
        OptimizeScaleValues: true
           PersistentMemory: false
```

See Also   convert | dfilt | tf2sos

Purpose   Fixed-point scaling modes in direct-form FIR filter

Syntax   specifyall(hd)
specifyall(hd,false)
specifyall(hd,true)
specifyall(hs)
specifyall(hs,false)

Description   specifyall sets all of the autoscale property values of direct-form
FIR filters to false and all *modes of the filters to SpecifyPrecision.
In this table, you see the results of using specifyall with direct-form
FIR filters.

Property Name	Default	Setting After Applying specifyall
CoeffAutoScale	true	false
OutputMode	AvoidOverflow	SpecifyPrecision
ProductMode	FullPrecision	SpecifyPrecision
AccumMode	KeepMSB	SpecifyPrecision
RoundMode	convergent	convergent
OverflowMode	wrap	wrap

specifyall(hd) gives you maximum control over all settings in a filter
hd by setting all of the autoscale options that are true to false, turning
off all autoscaling and resetting all modes — OutputMode, ProductMode,
and AccumMode — to SpecifyPrecision. After you use specifyall,
you must supply the property values for the mode- and scaling related
properties.

specifyall provides an alternative to changing all these properties
individually. Using specifyall changes all of the settings. To set some
but not all of the modes, set each property as you require.

specifyall(hd,false) performs the opposite operation of
specifyall(hd) by setting all of the autoscale options to true; all

specifyall

of the modes to their default values; and hiding the fraction length properties in the display, meaning you cannot access them to set them or view them.

`specifyall(hd,true)` is equivalent to `specifyall(hd)`.

`specifyall(hs)` sets all the data type fixed-point properties of the filter System object `hs` to `'Custom'` so that you can easily specify all the fixed-point settings. If the object has a `FullPrecisionOverride` property, its value is set to `false`. `specifyall` is intended as a shortcut to changing all the fixed-point properties one by one.

`specifyall(hs,false)` sets all fixed-point properties of the filter System object `hs` to their default values and sets the filter to full-precision mode, if possible.

Examples This example demonstrates using `specifyall` to provide access to all of the fixed-point settings of an FIR filter implemented with the direct-form structure. Using `specifyall` disables all of the automatic filter scaling and reset the mode values.

```
b = fircband(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],{'w' 'c'});
hd = dfilt.dffir(b);
hd.arithmetic = 'fixed'
specifyall(hd)
```

The mode properties `InputMode`, `ProductMode`, and `AccumMode` have the value `SpecifyPrecision` and the fraction length properties appear in the display. Now you use the properties (`InputFracLength`, `ProdFracLength`, `AccumFracLength`) to set the precision the filter applies to the input, product, and accumulator operations. `CoeffAutoScale` switches to `false`, which means that the filter coefficients will not be scaled automatically to prevent overflows. None of the other filter properties change when you apply `specifyall`.

See Also double | reffilter

Purpose   Step response for filter

Syntax
```
[h,t] = stepz(hfilt)
[h,t] = stepz(hfilt,n)
[h,t] = stepz(hfilt,n,fs)
[h,t] = stepz(hfilt,[],fs)
stepz(hfilt)
[h,t] = stepz(hs)
[h,t] = stepz(hs,Name,Value)
stepz(hs)
```

Description   stepz returns the step response based on the current filter coefficients. This section describes common stepz operation with adaptive filters, discrete-time filters, multirate filters, and filter System objects. For more input options, refer to stepz in Signal Processing Toolbox documentation.

[h,t] = stepz(hfilt) returns the impulse response h and the corresponding time points w at which the step response of hfilt is computed. The step response is evaluated at 10 1-second intervals—(0:9)'.

[h,t] = stepz(hfilt,n) returns the step response evaluated at floor(n) 1-second intervals—(0:floor(n)-1)'.

[h,t] = stepz(hfilt,n,fs) returns the step response evaluated at floor(n) 1/fs-second intervals—(0:floor(n)-1)'/fs.

[h,t] = stepz(hfilt,[],fs) returns the step response evaluated at 10 1/fs-second intervals—(0:9)'/fs.

stepz(hfilt) uses FVTool to plot the step response of the filter. You can also provide the optional input arguments n and fs with this syntax.

[h,t] = stepz(hs) returns the step response for the filter System object hs. The impulse response is evaluated at 10 1-second intervals—(0:9)'. You can also provide the optional input arguments n and fs with this syntax.

[h,t] = stepz(hs,Name,Value) returns an step response with additional options specified by one or more Name,Value pair arguments.

stepz(hs) uses FVTool to plot the step response of the filter System object hs.

Note stepz works for both real and complex filters. When you omit the output arguments, stepz plots only the real part of the step response.

Input Arguments

hfilt

hfilt is either:

- An adaptive adaptfilt, discrete-time dfilt, or multirate mfilt filter object

- A vector of adaptive, discrete-time, or multirate filter objects

The multirate filter step response is computed relative to the rate at which the filter is running. When you specify fs (the sampling rate) as an input argument, stepz assumes the filter is running at that rate.

For multistage cascades, stepz forms a single-stage multirate filter that is equivalent to the cascade. It then computes the response relative to the rate at which the equivalent filter is running. stepz does not support all multistage cascades. The function analyzes only those cascades for which there exists an equivalent single-stage filter.

As an example, consider a 2-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an interpolation factor of 4. In this case, an equivalent single-stage filter exists with an overall interpolation factor of 8. stepz uses this equivalent filter for the analysis. If a sampling frequency fs is specified as an input argument to stepz, the function interprets fs as the rate at which the equivalent filter is running.

hs

Filter System object. The function supports the following filter structures.

- `dsp.FIRFilter`

- `dsp.BiquadFilter`

- `dsp.FIRInterpolator`

- `dsp.CICInterpolator`

- `dsp.FIRDecimator`

- `dsp.CICDecimator`

- `dsp.FIRRateConverter`

n

Number of samples.

> **Default:** 10

fs

Sampling frequency.

> **Default:** 1

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Arithmetic

`` `double' `` | `'single'` | `'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify

'fixed' , the arithmetic changes depending on the setting of the
CoefficientDataType property and whether the System object is
locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.
Locked	'Same as input'	When the input data type is 'double' or 'fixed', the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	'Custom'	The function performs fixed-point

System Object State	Coefficient Data Type	Rule
		analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

h

Complex, n-element step response vector. If `hfilt` is a vector of filters, h is a complex, `length(hfilt)`-by-n matrix of step response vectors corresponding to each filter in `hfilt`. If n is not specified, the function uses a default value of 8192.

For adaptive filters, h is the instantaneous step response.

t

Time vector of length n, in seconds. t consists of n points equally spaced from 0 to `floor(n)/fs`. If n is not specified, the function uses a default value of 10. If `fs` is not specified, the function uses a default value of 1.

See Also   freqz | impz

sysobj

Purpose   Create filter System object from discrete-time filter or multirate filter

Syntax   `hs = sysobj(hfilt)`

Description   `hs = sysobj(hfilt)` creates a new filter System object `hs` from the `dfilt` or `mfilt` object, `hfilt`.

The function supports a subset of `dfilt` and `mfilt` objects. The following table lists supported filter structures for `hfilt` and the filter System object that the function creates.

Single-rate or multirate filter	Filter System object
Lattice AR(`dfilt.latticear`)	`dsp.AllpoleFilter`
Direct Form I (`dfilt.df1`)	`dsp.IIRFilter`
Direct Form I transposed (`dfilt.df1t`)	`dsp.IIRFilter`
Direct Form II (`dfilt.df2`)	`dsp.IIRFilter`
Direct Form II transposed (`dfilt.df2t`)	`dsp.IIRFilter`
Direct-form FIR (`dfilt.dffir`)	`dsp.FIRFilter`
Direct-form FIR transposed (`dfilt.dffirt`)	`dsp.FIRFilter`
Direct-form symmetric FIR (`dfilt.dfsymfir`)	`dsp.FIRFilter`
Direct-form antisymmetric FIR (`dfilt.dfasymfir`)	`dsp.FIRFilter`
Discrete-time, lattice, moving-average (`dfilt.latticemamin`)	`dsp.FIRFilter`
Discrete-time, second-order section, direct-form I (`dfilt.df1sos`)	`dsp.BiquadFilter`

Single-rate or multirate filter	Filter System object
Discrete-time, second-order section, direct-form I transposed (`dfilt.df1tsos`)	`dsp.BiquadFilter`
Discrete-time, second-order section, direct-form II (`dfilt.df2sos`)	`dsp.BiquadFilter`
Discrete-time, second-order section, direct-form II transposed (`dfilt.df2tsos`)	`dsp.BiquadFilter`
Direct-form FIR polyphase decimator (`mfilt.firdecim`)	`dsp.FIRDecimator`
Direct-form transposed FIR polyphase decimator (`mfilt.firtdecim`)	`dsp.FIRDecimator`
Fixed-point CIC decimator (`mfilt.cicdecim`)	`dsp.CICDecimator`
FIR polyphase interpolator (`mfilt.firinterp`)	`dsp.FIRInterpolator`
FIR linear interpolator (`mfilt.linearinterp`)	`dsp.FIRInterpolator`
Fixed-point CIC interpolator (`mfilt.cicinterp`)	`dsp.CICInterpolator`
Direct-form FIR polyphase sample rate converter (`mfilt.firsrc`)	`dsp.FIRRateConverter`

Input Arguments

hfilt

Discrete-time filter (`dfilt`) or multirate filter (`mfilt`) object. The preceding table lists supported filter structures.

sysobj

The function interprets the `PersistentMemory` property differently for discrete-time and multirate filters:

- If `hfilt` is a discrete-time filter with the `PersistentMemory` property set to `true`, then the filter states are copied into the initial conditions properties of `hs`. Otherwise, initial conditions are ignored.

- Multirate-filter System objects do not have initial conditions. If `hfilt` is a multirate filter with the `PersistentMemory` property set to `true`, the initial conditions are not copied into `hs`, and the function issues a warning.

The function does not support some properties for SOS filter structures:

- If the `CastBeforeSum` property is set to `false`, the function issues a warning. `dsp.BiquadFilter` System objects always have a cast before a sum.

- If the `Signed` property is `false`, the function issues an error. `dsp.BiquadFilter` System objects do not support unsigned arithmetic.

Output Arguments

hs

Filter System object. The function maps almost all properties of `hfilt` into the filter System object. However, some properties are not mapped exactly:

- Filter System objects do not have a `CoeffAutoScale` property. The function specifies a word length and a fraction length regardless of whether the `CoeffAutoScale` property of `hfilt` is `true` or `false`.

- `dsp.BiquadFilter` System objects do not have a `FullPrecisionOverride` property. Full-precision values in `hfilt` are mapped to word and fraction lengths in `hs`. These settings correspond to the full-precision setting of the input data type.

Examples

Convert a discrete-time filter object to a System object:

```
hfilt = dfilt.df1sos; %Direct-form I SOS
hs = sysobj(hfilt);    %Biquadratic IIR filter
```

See Also block

tf2ca

Purpose   Transfer function to coupled allpass

Syntax   ```
[d1,d2] = tf2ca(b,a)
[d1,d2] = tf2ca(b,a)
```

Description   `[d1,d2] = tf2ca(b,a)` where `b` is a real, symmetric vector of
numerator coefficients and `a` is a real vector of denominator coefficients,
corresponding to a stable digital filter, returns real vectors `d1` and `d2`
containing the denominator coefficients of the allpass filters *H1(z)* and
H2(z) such that

$$H(z) = \frac{B(z)}{A(z)} = \frac{1}{2[H1(z) + H2(z)]}$$

representing a coupled allpass decomposition.

`[d1,d2] = tf2ca(b,a)` where `b` is a real, antisymmetric vector of
numerator coefficients and `a` is a real vector of denominator coefficients,
corresponding to a stable digital filter, returns real vectors `d1` and `d2`
containing the denominator coefficients of the allpass filters *H1(z)* and
H2(z) such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)[H1(z) - H2(z)]$$

In some cases, the decomposition is not possible with real *H1(z)* and
H2(z). In those cases a generalized coupled allpass decomposition may
be possible, whose syntax is

```
[d1,d2,beta] = tf2ca(b,a)
```

to return complex vectors `d1` and `d2` containing the denominator
coefficients of the allpass filters *H1(z)* and *H2(z)*, and a complex scalar
`beta`, satisfying `|beta| = 1`, such that

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)\left[\bar{\beta} \cdot H1(z) + \beta \cdot H2(z)\right]$$

representing the generalized allpass decomposition.

In the above equations, *H1(z)* and *H2(z)* are real or complex allpass IIR filters given by

$$H1(z) = \frac{fliplr(\overline{(D1(z))})}{D1(z)}, H2(1)(z) = \frac{fliplr(\overline{(D2(1)(z))})}{D2(1)(z)}$$

where *D1(z)* and *D2(z)* are polynomials whose coefficients are given by d1 and d2.

Note A coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to Signal Processing Toolbox User's Guide.

Examples

```
[b,a]=cheby1(9,.5,.4);
[d1,d2]=tf2ca(b,a);  % TF2CA returns denominators of the allpass.
num = 0.5*conv(fliplr(d1),d2)+0.5*conv(fliplr(d2),d1);
den = conv(d1,d2);  % Reconstruct numerator and denonimator.
MaxDiff=max([max(b-num),max(a-den)]); % Compare original and reconstructed
        % numerator and denominators.
```

See Also   ca2tf | cl2tf | iirpowcomp | latc2tf | tf2latc

tf2cl

Purpose   Transfer function to coupled allpass lattice

Syntax   ```
[k1,k2] = tf2cl(b,a)
[k1,k2] = tf2cl(b,a)
```

**Description**    `[k1,k2] = tf2cl(b,a)` where `b` is a real, symmetric vector of
                   numerator coefficients and `a` is a real vector of denominator coefficients,
                   corresponding to a stable digital filter, will perform the coupled allpass
                   decomposition

$$H(z) = \frac{B(z)}{A(z)} = \frac{1}{2\left[H1(z) + H2(z)\right]}$$

of a stable IIR filter *H(z)* and convert the allpass transfer functions
*H1(z)* and *H2(z)* to a coupled lattice allpass structure with coefficients
given in vectors `k1` and `k2`.

`[k1,k2] = tf2cl(b,a)` where `b` is a real, antisymmetric vector of
numerator coefficients and `a` is a real vector of denominator coefficients,
corresponding to a stable digital filter, performs the coupled allpass
decomposition

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)\left[H1(z) - H2(z)\right]$$

of a stable IIR filter *H(z)* and converts the allpass transfer functions
*H1(z)* and *H2(z)* to a coupled lattice allpass structure with coefficients
given in vectors `k1` and `k2`.

In some cases, the decomposition is not possible with real *H1(z)* and
*H2(z)*. In those cases, a generalized coupled allpass decomposition may
be possible, using the command syntax

```
[k1,k2,beta] = tf2cl(b,a)
```

to perform the generalized allpass decomposition of a stable IIR filter
H(z) and convert the complex allpass transfer functions *H1(z)* and *H2(z)*
to corresponding lattice allpass filters

$$H(z) = \frac{B(z)}{A(z)} = \left(\frac{1}{2}\right)\left[\overline{\beta} \cdot H1(z) + \beta \cdot H2(z)\right]$$

where `beta` is a complex scalar of magnitude equal to 1.

Note Coupled allpass decomposition is not always possible. Nevertheless, Butterworth, Chebyshev, and Elliptic IIR filters, among others, can be factored in this manner. For details, refer to Signal Processing Toolbox User's Guide.

Examples

```
[b,a]=cheby1(9,.5,.4);
[k1,k2]=tf2cl(b,a); % Get the reflection coeffs. for the lattices.
[num1,den1]=latc2tf(k1,'allpass'); % Convert each allpass lattice
[num2,den2]=latc2tf(k2,'allpass'); % back to transfer function.
num = 0.5*conv(num1,den2)+0.5*conv(num2,den1);
den = conv(den1,den2); % Reconstruct numerator and denonimator.
MaxDiff=max([max(b-num),max(a-den)]); % Compare original and reconstructed
        % numerator and denominators.
```

See Also ca2tf | cl2tf | iirpowcomp | latc2tf | tf2ca | tf2latc

validstructures

Purpose	Structures for specification object with design method

Syntax	`filtstruct = validstructures(D)`
	`C = validstructures(D,METHOD)`
	`Cs = validstructures(D,...,'SystemObject',sysobjflag)`

Description	`filtstruct = validstructures(D)` returns a structure array containing all valid filter structures for the filter specification object, `D`, organized by design method. Each design method is a field in the structure array, `filtstruct`. The fields contain a cell array of strings. `C = validstructures(D,METHOD)` returns the valid structures for the filter specification object, `D`, and the design method, `METHOD`, in a cell array of strings. `Cs = validstructures(D,...,'SystemObject',sysobjflag)` returns the valid structures for designing a filter System object when sysobjflag is `true`. To use System objects, you must have the DSP System Toolbox product installed. When sysobjflag is `false`, the function returns valid structures for designing `dfilt` and `mfilt` objects, as described previously. Design methods and design options for filter System objects are not necessarily the same as those for `dfilt` and `mfilt` objects.

Examples	Design a default lowpass filter specification object. Return all valid design methods and structures in a structure array. Display the fieldnames to see all valid design methods. Display the valid filter structures for the `equiripple` field.
	
	`D = fdesign.lowpass;`
	`filtstruct = validstructures(D);`
	`fieldnames(filtstruct)`
	`filtstruct.equiripple`

Create a highpass filter of order 50 with a 3-dB frequency of 0.2. Obtain the available structures for a Butterworth design.

```
D = fdesign.highpass('N,F3dB',50,0.2);
C = validstructures(D,'butter');
```

If you have DSP System Toolbox software installed, use the
'SystemObject',*sysobjflag* syntax to return valid structures for a
filter System object:

```
Cs = validstructures(D,'butter','SystemObject',true);
```

See Also   design | designmethods | designopts | fdesign

window

Purpose	FIR filter using windowed impulse response
Syntax	`h = window(d,fcnhndl,fcnarg)`
	`h = window(d,win)`

description   `h = window(d,fcnhndl,fcnarg)` designs an FIR filter using the specifications in filter specification object `d`. Depending on the specification type of `d`, the returned filter is either a single-rate digital filter — a `dfilt`, or a multirate digital filter — an `mfilt`.

`fcnhndl` is a handle to a filter design function that returns a window vector, such as the `hamming` or `blackman` functions. `fcnarg` is an optional argument that returns a window. You pass the function to `window`. Refer to example 1 in the following section to see the function argument used to design the filter.

`h = window(d,win)` designs a filter using the vector you supply in `win`. The length of vector `win` must be the same as the impulse response of the filter, which is equal to the filter order plus one. Example 2 shows this being done.

Examples   These examples design filters using the two design techniques of specifying a function handle or passing a window vector as an input argument.

Example 1

Use a function handle and optional input arguments to design a multirate filter. We use a function handle to `hamming` to provide the window. Since this example creates a decimator filter specifications object, `window` returns a multirate filter.

```
d = fdesign.decimator(4,'lowpass','N,Fc',30,1000,10000);
% Lowpass decimator with a 6-dB down frequency of 1 kHz
% Order equal to 30 and sampling frequency 10 kHz
Hd =window(d,'window',@hamming);
fvtool(Hd)
```

Example 2

Use a window vector provided by the kaiser window function to design a Nyquist filter. The window length must be the filter order plus one. .

```
d = fdesign.nyquist(5,'n',150);
% Kaiser window with beta parameter 2.5
Hd = window(d,'window',kaiser(151,2.5));
fvtool(Hd)
```

window

See Also   firls | kaiserwin

Purpose   Zero-phase response for filter

Syntax
```
[hr,w] = zerophase(hfilt)
[hr,w] = zerophase(hfilt,n)
zerophase(hfilt)
[hr,w] = zerophase(hs)
[hr,w] = zerophase(hs,n)
[hr,w] = zerophase(hs,Name,Value)
zerophase(hs)
```

Description   zerophase returns the zero-phase response based on the current filter coefficients. This section describes common zerophase operation with adaptive, discrete-time, and multirate filters. For more input options, refer to zerophase in Signal Processing Toolbox documentation.

[hr,w] = zerophase(hfilt) returns the instantaneous zero-phase response hr and the frequencies w in radians at which zerophase evaluated the response. The zero-phase response is evaluated at 8192 points equally spaced around the upper half of the unit circle.

[hr,w] = zerophase(hfilt,n) returns the instantaneous zero-phase response hr and the frequencies w in radians at which zerophase evaluated the response. The zero-phase response is evaluated at n points equally spaced around the upper half of the unit circle.

zerophase(hfilt) displays the zero-phase response of ha in the Filter Visualization Tool (FVTool).

[hr,w] = zerophase(hs) returns a zero-phase response for the filter System object hs using 8192 samples.

[hr,w] = zerophase(hs,n) returns a zero-phase response for the filter System object hs using n samples.

[hr,w] = zerophase(hs,Name,Value) returns a zero-phase response with additional options specified by one or more Name,Value pair arguments.

zerophase(hs) uses FVTool to plot the zero-phase response of the filter System object hs.

zerophase

**Input
Arguments**

hfilt

`hfilt` is either:

- An adaptive `adaptfilt`, discrete-time `dfilt`, or multirate `mfilt` filter object

- A vector of adaptive, discrete-time, or multirate filter objects

The multirate filter zero-phase response is computed relative to the rate at which the filter is running. When you specify `fs` (the sampling rate) as an input argument, `zerophase` assumes the filter is running at that rate.

For multistage cascades, `zerophase` forms a single-stage multirate filter that is equivalent to the cascade. It then computes the response relative to the rate at which the equivalent filter is running. `zerophase` does not support all multistage cascades. The function analyzes only those cascades for which there exists an equivalent single-stage filter.

As an example, consider a 2-stage interpolator where the first stage has an interpolation factor of 2 and the second stage has an interpolation factor of 4. In this case, an equivalent single-stage filter exists with an overall interpolation factor of 8. `zerophase` uses this equivalent filter for the analysis. If a sampling frequency `fs` is specified as an input argument to `zerophase`, the function interprets `fs` as the rate at which the equivalent filter is running.

hs

Filter System object. The function supports the following filter structures.

- `dsp.FIRFilter`

- `dsp.BiquadFilter`

- `dsp.FIRInterpolator`

- `dsp.CICInterpolator`

- `dsp.FIRDecimator`

- `dsp.CICDecimator`

- `dsp.FIRRateConverter`

n

Number of samples. For an FIR filter where `n` is a power of two, the computation is done faster using FFTs.

> **Default:** 8192

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments, where `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Arithmetic
`'double' | 'single' | 'fixed'`

For filter System object inputs only, specify the arithmetic used during analysis. When you specify `'double'` or `'single'`, the function performs double- or single-precision analysis. When you specify `'fixed'`, the arithmetic changes depending on the setting of the `CoefficientDataType` property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	`'Same as input'`	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

hr

Complex, n-element zero-phase response vector. If hfilt is a vector of filters, hr is a complex, length(hfilt)-by-n matrix of phase response vectors whose columns correspond to each filter in hfilt. If n is not specified, the function uses a default value of 8192.

For adaptive filters, hr is the instantaneous zero-phase response.

w

Frequency vector of length n, in radians/sample. w consists of n points equally spaced around the upper half of the unit circle (from 0 to π radians/sample). If hfilt is a vector of filters, w is a complex, length(hfilt)-by-n matrix of phase response vectors whose columns correspond to each filter in hfilt. If n is not specified, the function uses a default value of 8192.

See Also   freqz | fvtool | grpdelay | impz | mfilt | phasez | zerophase | zplane

zpkbpc2bpc

Purpose   Zero-pole-gain complex bandpass frequency transformation

Syntax   `[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)`

Description `[Z2,P2,K2,AllpassNum,AllpassDen] = zpkbpc2bpc(Z,P,K,Wo,Wt)`
returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
transformed from the complex bandpass prototype by applying
a first-order complex bandpass to complex bandpass frequency
transformation.

It also returns the numerator, `AllpassNum`, and the denominator,
`AllpassDen`, of the allpass mapping filter. The original lowpass filter is
given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places two features of an original
filter, located at frequencies W_{o1} and W_{o2}, at the required target
frequency locations, W_{t1}, and W_{t2} respectively. It is assumed that W_{t2}
is greater than W_{t1}. In most of the cases the features selected for the
transformation are the band edges of the filter passbands. In general
it is possible to select any feature; e.g., the stopband edge, the DC, the
deep minimum in the stopband, or other ones.

Relative positions of other features of an original filter do not change in
the target filter. This means that it is possible to select two features of
an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
precede F_2 after the transformation. However, the distance between F_1
and F_2 will not be the same before and after the transformation.

This transformation can also be used for transforming other types of
filters; e.g., complex notch filters or resonators can be repositioned at
two distinct desired frequencies at any place around the unit circle;
e.g., in the adaptive system.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
approach:

`[b, a] = ellip(3,0.1,30,0.409);`

Create a complex passband from 0.25 to 0.75:

```
[b, a] = iirlp2bpc(b,a,0.5,[0.25,0.75]);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpkbpc2bpc(z,p,k,[0.25, 0.75],[-0.75, -0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Comparing the filters in FVTool shows the example results. Use the features in FVTool to check the filter coefficients, or other filter analyses.

zpkbpc2bpc

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also   zpkftransf | allpassbpc2bpc | iirbpc2bpc

Purpose   Zero-pole-gain frequency transformation

Syntax   [Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen)

Description   [Z2,P2,K2] = zpkftransf(Z,P,K,AllpassNum,AllpassDen) returns
                       zeros, Z_2, poles, P_2, and gain factor, K_2, of the transformed lowpass
                       digital filter. The prototype lowpass filter is given with zeros, Z, poles,
                       P, and gain factor, K. If AllpassDen is not specified it will default to 1.
                       If neither AllpassNum nor AllpassDen is specified, then the function
                       returns the input filter.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
                       approach:

```
[b, a] = ellip(3,0.1,30,0.409);
[AlpNum, AlpDen] = allpasslp2lp(0.5, 0.25);
[z2, p2, k2] = zpkftransf(roots(b),roots(a),b(1),AlpNum,AlpDen);
```

                       Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

                       After transforming the filter, you get the response shown in the figure,
                       where the passband has been shifted towards zero.

zpkftransf

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
FTFNum	Numerator of the mapping filter
FTFDen	Denominator of the mapping filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter

See Also   iirftransf

Purpose   Zero-pole-gain lowpass to bandpass frequency transformation

Syntax   [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bp(Z,P,K,Wo,Wt)

Description   [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bp(Z,P,K,Wo,Wt)
                   returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
                   transformed from the real lowpass prototype by applying a second-order
                   real lowpass to real bandpass frequency mapping.

                   It also returns the numerator, AllpassNum, and the denominator
                   AllpassDen, of the allpass mapping filter. The prototype lowpass filter
                   is given with zeros, Z, poles, P, and gain factor, K.

                   This transformation effectively places one feature of an original filter,
                   located at frequency -W_o, at the required target frequency location,
                   W_{t1}, and the second feature, originally at +W_o, at the new location,
                   W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation
                   implements the "DC Mobility," which means that the Nyquist feature
                   stays at Nyquist, but the DC feature moves to a location dependent
                   on the selection of W_t.

                   Relative positions of other features of an original filter do not change in
                   the target filter. This means that it is possible to select two features of
                   an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
                   precede F_2 after the transformation. However, the distance between F_1
                   and F_2 will not be the same before and after the transformation.

                   Choice of the feature subject to the lowpass to bandpass transformation
                   is not restricted only to the cutoff frequency of an original lowpass filter.
                   In general it is possible to select any feature; e.g., the stopband edge,
                   the DC, the deep minimum in the stopband, or other ones.

                   Real lowpass to bandpass transformation can also be used for
                   transforming other types of filters; e.g., real notch filters or resonators
                   can be easily doubled and positioned at two distinct, desired frequencies.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
                   approach:

```
[B,A] = ellip(3,0.1,30,0.409);
Z = roots(B);
P = roots(A);
K = B(1);
[Z2,P2,K2] = zpklp2bp(Z,P,K, 0.5, [0.2 0.3]);
hfvt = fvtool(B,A,K2*poly(Z2),poly(P2));
legend(hfvt,'Prototype Lowpass Filter', 'Bandpass Filter');
axis([0 1 -70 10]);
```


Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also   zpkftransf | allpasslp2bp | iirlp2bp

zpklp2bpc

Purpose   Zero-pole-gain lowpass to complex bandpass frequency transformation

Syntax   `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bpc(Z,P,K,Wo,Wt)`

Description `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bpc(Z,P,K,Wo,Wt)`
returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
transformed from the real lowpass prototype by applying a first-order
real lowpass to complex bandpass frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator,
`AllpassDen`, of the allpass mapping filter. The prototype lowpass filter
is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter,
located at frequency $-W_o$, at the required target frequency location, W_{t1},
and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is
assumed that W_{t2} is greater than W_{t1}.

Relative positions of other features of an original filter do not change in
the target filter. This means that it is possible to select two features of
an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
precede F_2 after the transformation. However, the distance between F_1
and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandpass transformation
is not restricted only to the cutoff frequency of an original lowpass filter.
In general it is possible to select any feature; e.g., the stopband edge,
the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming
other types of filters; e.g., real notch filters or resonators can be doubled
and positioned at two distinct desired frequencies at any place around
the unit circle forming a pair of complex notches/resonators. This
transformation can be used for designing bandpass filters for radio
receivers from the high-quality prototype lowpass filter.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bpc(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also   zpkftransf | allpasslp2bpc | iirlp2bpc

zpklp2bs

Purpose   Zero-pole-gain lowpass to bandstop frequency transformation

Syntax   `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bs(Z,P,K,Wo,Wt)`

Description   `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bs(Z,P,K,Wo,Wt)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter transformed from the real lowpass prototype by applying a second-order real lowpass to real bandstop frequency mapping.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

This transformation effectively places one feature of an original filter, located at frequency $-W_o$, at the required target frequency location, W_{t1}, and the second feature, originally at $+W_o$, at the new location, W_{t2}. It is assumed that W_{t2} is greater than W_{t1}. This transformation implements the "Nyquist Mobility," which means that the DC feature stays at DC, but the Nyquist feature moves to a location dependent on the selection of W_o and W_ts.

Relative positions of other features of an original filter change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. After the transformation feature F_2 will precede F_1 in the target filter. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation is not restricted only to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones.

Examples   Design a prototype real IIR halfband filter using a standard elliptic approach:

```
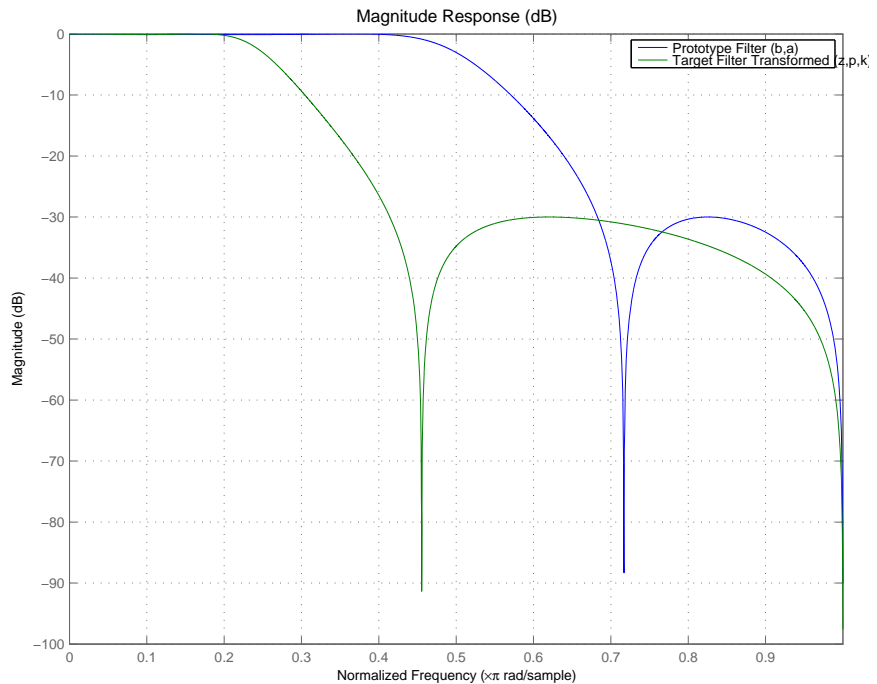[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
```

```
k = b(1);
[z2,p2,k2] = zpklp2bs(z, p, k, 0.5, [0.2 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References Constantinides, A.G., "Spectral transformations for digital filters," *IEEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

zpklp2bs

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Design of bandpass digital filters," *IEEE Proceedings*, vol. 1, pp. 1129-1231, June 1969.

See Also   zpkftransf | allpasslp2bs | iirlp2bs

Purpose   Zero-pole-gain lowpass to complex bandstop frequency transformation

Syntax   [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bsc(Z,P,K,Wo,Wt)

Description [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2bsc(Z,P,K,Wo,Wt)
returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
transformed from the real lowpass prototype by applying a first-order
real lowpass to complex bandstop frequency transformation.

It also returns the numerator, AllpassNum, and the denominator,
AllpassDen, of the allpass mapping filter. The prototype lowpass filter
is given with zeros, Z, poles, P, and gain factor, K.

This transformation effectively places one feature of an original filter,
located at frequency -W_o, at the required target frequency location, W_{t1},
and the second feature, originally at +W_o, at the new location, W_{t2}. It is
assumed that W_{t2} is greater than W_{t1}. Additionally the transformation
swaps passbands with stopbands in the target filter.

Relative positions of other features of an original filter do not change in
the target filter. This means that it is possible to select two features of
an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
precede F_2 after the transformation. However, the distance between F_1
and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to bandstop transformation
is not restricted only to the cutoff frequency of an original lowpass filter.
In general it is possible to select any feature; e.g., the stopband edge,
the DC, the deep minimum in the stopband, or other ones.

Lowpass to bandpass transformation can also be used for transforming
other types of filters; e.g., real notch filters or resonators can be doubled
and positioned at two distinct desired frequencies at any place around
the unit circle forming a pair of complex notches/resonators.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
approach:

```
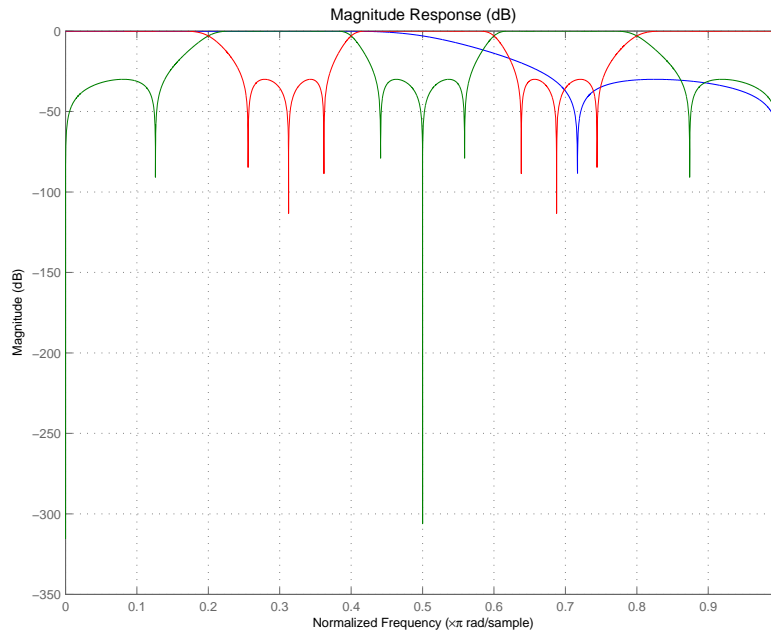[b, a] = ellip(3,0.1,30,0.409);
```

```
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2bsc(z, p, k, 0.5, [0.2, 0.3]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also   zpkftransf | allpasslp2bsc | iirlp2bsc

Purpose   Zero-pole-gain lowpass to highpass frequency transformation

Syntax   [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2hp(Z,P,K,Wo,Wt)

Description   [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2hp(Z,P,K,Wo,Wt)
                     returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
                     transformed from the real lowpass prototype by applying a first-order
                     real lowpass to real highpass frequency mapping. This transformation
                     effectively places one feature of an original filter, located at frequency
                     W_o, at the required target frequency location, W_t, at the same time
                     rotating the whole frequency response by half of the sampling frequency.
                     Result is that the DC and Nyquist features swap places.

                     It also returns the numerator, AllpassNum, and the denominator,
                     AllpassDen, of the allpass mapping filter. The prototype lowpass filter
                     is given with zeros, Z, poles, P, and the gain factor, K.

                     Relative positions of other features of an original filter change in the
                     target filter. This means that it is possible to select two features
                     of an original filter, F_1 and F_2, with F_1 preceding F_2. After the
                     transformation feature F_2 will precede F_1 in the target filter. However,
                     the distance between F_1 and F_2 will not be the same before and after
                     the transformation.

                     Choice of the feature subject to the lowpass to highpass transformation
                     is not restricted to the cutoff frequency of an original lowpass filter. In
                     general it is possible to select any feature; e.g., the stopband edge, the
                     DC, or the deep minimum in the stopband, or other ones.

                     Lowpass to highpass transformation can also be used for transforming
                     other types of filters; e.g., notch filters or resonators can change their
                     position in a simple way without designing them again.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
                     approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
```

```
k = b(1);
[z2,p2,k2] = zpklp2hp(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

See Also   zpkftransf | allpasslp2hp | iirlp2hp

zpklp2lp

Purpose   Zero-pole-gain lowpass to lowpass frequency transformation

Syntax   `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2lp(Z,P,K,Wo,Wt)`

Description   `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2lp(Z,P,K,Wo,Wt)`
returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
transformed from the real lowpass prototype by applying a first-order
real lowpass to real lowpass frequency mapping. This transformation
effectively places one feature of an original filter, located at frequency
W_o, at the required target frequency location, W_t.

It also returns the numerator, `AllpassNum`, and the denominator,
`AllpassDen`, of the allpass mapping filter. The prototype lowpass filter
is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Relative positions of other features of an original filter do not change in
the target filter. This means that it is possible to select two features of
an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
precede F_2 after the transformation. However, the distance between F_1
and F_2 will not be the same before and after the transformation.

Choice of the feature subject to the lowpass to lowpass transformation
is not restricted to the cutoff frequency of an original lowpass filter. In
general it is possible to select any feature; e.g., the stopband edge, the
DC, the deep minimum in the stopband, or other ones.

Lowpass to lowpass transformation can also be used for transforming
other types of filters; e.g., notch filters or resonators can change their
position in a simple way without designing them again.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
approach:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2lp(z, p, k, 0.5, 0.25);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Using zpklp2lp creates the desired half band IIR filter with the transformed features that you specify in the transformation function. This figure shows the results.

Magnitude Response (dB)

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter

zpklp2lp

Variable	Description
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Constantinides, A.G., "Spectral transformations for digital filters," *IEE Proceedings*, vol. 117, no. 8, pp. 1585-1590, August 1970.

Nowrouzian, B. and A.G. Constantinides, "Prototype reference transfer function parameters in the discrete-time frequency transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, vol. 2, pp. 1078-1082, August 1990.

Nowrouzian, B. and L.T. Bruton, "Closed-form solutions for discrete-time elliptic transfer functions," *Proceedings of the 35th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 784-787, 1992.

Constantinides, A.G., "Frequency transformations for digital filters," *Electronics Letters*, vol. 3, no. 11, pp. 487-489, November 1967.

See Also   zpkftransf | allpasslp2lp | iirlp2lp

Purpose   Zero-pole-gain lowpass to M-band frequency transformation

Syntax   [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt)
                 [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt,Pass)

Description [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2mb(Z,P,K,Wo,Wt)
                 returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
                 transformed from the real lowpass prototype by applying an Mth-order
                 real lowpass to real multibandpass frequency mapping. By default the
                 DC feature is kept at its original location.

                 [Z2,P2,K2,AllpassNum,AllpassDen] =
                 zpklp2mb(Z,P,K,Wo,Wt,Pass) allows you to specify an additional
                 parameter, Pass, which chooses between using the "DC Mobility" and
                 the "Nyquist Mobility". In the first case the Nyquist feature stays at its
                 original location and the DC feature is free to move. In the second
                 case the DC feature is kept at an original frequency and the Nyquist
                 feature is allowed to move.

                 It also returns the numerator, AllpassNum, and the denominator,
                 AllpassDen, of the allpass mapping filter. The prototype lowpass filter
                 is given with zeros, Z, poles, P, and gain factor, K.

                 This transformation effectively places one feature of an original filter,
                 located at frequency W_o, at the required target frequency locations,
                 $W_{t1},...,W_{tM}$.

                 Relative positions of other features of an original filter do not change in
                 the target filter. This means that it is possible to select two features of
                 an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
                 precede F_2 after the transformation. However, the distance between F_1
                 and F_2 will not be the same before and after the transformation.

                 Choice of the feature subject to this transformation is not restricted
                 to the cutoff frequency of an original lowpass filter. In general it is
                 possible to select any feature; e.g., the stopband edge, the DC, the deep
                 minimum in the stopband, or other ones.

                 This transformation can also be used for transforming other types of
                 filters; e.g., notch filters or resonators can be easily replicated at a

number of required frequency locations. A good application would be
an adaptive tone cancellation circuit reacting to the changing number
and location of tones.

Examples Design a prototype real IIR halfband filter using a standard elliptic
approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z1,p1,k1] = zpklp2mb(z, p, k, 0.5, [2 4 6 8]/10, 'pass');
[z2,p2,k2] = zpklp2mb(z, p, k, 0.5, [2 4 6 8]/10, 'stop');
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k1*poly(z1), poly(p1), k2*poly(z2), poly(p2));
```

The resulting multiband filter that replicates features from the
prototype appears in the figure shown. Note the accuracy of the
replication process.

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter
Wt	Desired frequency location in the transformed target filter
Pass	Choice (`'pass'`/`'stop'`) of passband/stopband at DC, `'pass'` being the default
Z2	Zeros of the target filter

Variable	Description
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Franchitti, J.C., "All-pass filter interpolation and frequency transformation problems," *MSc Thesis*, Dept. of Electrical and Computer Engineering, University of Colorado, 1985.

Feyh, G., J.C. Franchitti and C.T. Mullis, "All-pass filter interpolation and frequency transformation problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

Mullis, C.T. and R.A. Roberts, *Digital Signal Processing*, section 6.7, Reading, Massachusetts, Addison-Wesley, 1987.

Feyh, G., W.B. Jones and C.T. Mullis, *An extension of the Schur Algorithm for frequency transformations, Linear Circuits, Systems and Signal Processing: Theory and Application*, C. J. Byrnes et al Eds, Amsterdam: Elsevier, 1988.

See Also   zpkftransf | allpasslp2mb | iirlp2mb

Purpose   Zero-pole-gain lowpass to complex M-band frequency transformation

Syntax   `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklpmbc(Z,P,K,Wo,Wt)`

Description   `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklpmbc(Z,P,K,Wo,Wt)`
                     returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
                     transformed from the real lowpass prototype by applying an Mth-order
                     real lowpass to complex multibandpass frequency transformation.

                     It also returns the numerator, `AllpassNum`, and the denominator,
                     `AllpassDen`, of the allpass mapping filter. The prototype lowpass filter
                     is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

                     This transformation effectively places one feature of an original filter,
                     located at frequency W_o, at the required target frequency locations,
                     $W_{t1},...,W_{tM}$.

                     Choice of the feature subject to this transformation is not restricted
                     to the cutoff frequency of an original lowpass filter. In general it is
                     possible to select any feature, for example, the stopband edge, the DC,
                     the deep minimum in the stopband, or other ones.

                     Relative positions of other features of an original filter do not change in
                     the target filter. This means that it is possible to select two features of
                     an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
                     precede F_2 after the transformation. However, the distance between F_1
                     and F_2 will not be the same before and after the transformation.

                     This transformation can also be used for transforming other types of
                     filters; e.g., to replicate notch filters and resonators at any required
                     location.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
                     approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

```
[z1,p1,k1] = zpklp2mbc(z, p, k, 0.5, [2 4 6 8]/10);
[z2,p2,k2] = zpklp2mbc(z, p, k, 0.5, [2 4 6 8]/10);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k1*poly(z1), poly(p1), k2*poly(z2), poly(p2));
```

You could review the coefficients to compare the filters, but the graphical comparison shown here is quicker and easier.

However, looking at the coefficients in FVTool shows the complex nature desired.

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter. It should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also   zpkftransf | allpasslp2mbc | iirlp2mbc

zpklp2xc

Purpose   Zero-pole-gain lowpass to complex N-point frequency transformation

Syntax   `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xc(Z,P,K,Wo,Wt)`

Description   `[Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xc(Z,P,K,Wo,Wt)`
returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
transformed from the real lowpass prototype by applying an `Nth`-order
real lowpass to complex multipoint frequency transformation.

It also returns the numerator, `AllpassNum`, and the denominator,
`AllpassDen`, of the allpass mapping filter. The prototype lowpass filter
is given with zeros, `Z`, poles, `P`, and gain factor, `K`.

Parameter `N` also specifies the number of replicas of the prototype
filter created around the unit circle after the transformation. This
transformation effectively places `N` features of an original filter, located
at frequencies $W_{o1},...,W_{oN}$, at the required target frequency locations,
$W_{t1},...,W_{tM}$.

Relative positions of other features of an original filter are the same
in the target filter for the Nyquist mobility and are reversed for the
DC mobility. For the Nyquist mobility this means that it is possible to
select two features of an original filter, F_1 and F_2, with F_1 preceding F_2.
Feature F_1 will still precede F_2 after the transformation. However,
the distance between F_1 and F_2 will not be the same before and after
the transformation. For DC mobility feature F_2 will precede F_1 after
the transformation.

Choice of the feature subject to this transformation is not restricted
to the cutoff frequency of an original lowpass filter. In general it is
possible to select any feature; e.g., the stopband edge, the DC, the deep
minimum in the stopband, or other ones. The only condition is that the
features must be selected in such a way that when creating `N` bands
around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of
filters; e.g., notch filters or resonators can be easily replicated at a
number of required frequency locations. A good application would be

an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples   Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2xc(z, p, k, [-0.5 0.5], [-0.25 0.25]);
```

Verify the result by comparing the prototype filter with the target filter:

```
fvtool(b, a, k2*poly(z2), poly(p2));
```

Plotting the filters on the same axes lets you compare the results graphically, shown here.

zpklp2xc

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency values to be transformed from the prototype filter. They should be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.
Wt	Desired frequency locations in the transformed target filter. They should be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

See Also   zpkftransf | allpasslp2xc | iirlp2xc

Purpose   Zero-pole-gain lowpass to N-point frequency transformation

Syntax   [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt)
                [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt,Pass)

Description [Z2,P2,K2,AllpassNum,AllpassDen] = zpklp2xn(Z,P,K,Wo,Wt)
                returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
                transformed from the real lowpass prototype by applying an Nth-order
                real lowpass to real multipoint frequency transformation, where N is the
                number of features being mapped. By default the DC feature is kept at
                its original location.

                [Z2,P2,K2,AllpassNum,AllpassDen] =
                zpklp2xn(Z,P,K,Wo,Wt,Pass) allows you to specify an additional
                parameter, Pass, which chooses between using the "DC Mobility" and
                the "Nyquist Mobility". In the first case the Nyquist feature stays at its
                original location and the DC feature is free to move. In the second
                case the DC feature is kept at an original frequency and the Nyquist
                feature is allowed to move.

                It also returns the numerator, AllpassNum, and the denominator,
                AllpassDen, of the allpass mapping filter. The prototype lowpass filter
                is given with zeros, Z, poles, P, and gain factor, K.

                Parameter N also specifies the number of replicas of the prototype
                filter created around the unit circle after the transformation. This
                transformation effectively places N features of an original filter, located
                at frequencies W_{o1},...,W_{oN}, at the required target frequency locations,
                W_{t1},...,W_{tM}.

                Relative positions of other features of an original filter are the same
                in the target filter for the Nyquist mobility and are reversed for the
                DC mobility. For the Nyquist mobility this means that it is possible to
                select two features of an original filter, F_1 and F_2, with F_1 preceding F_2.
                Feature F_1 will still precede F_2 after the transformation. However,
                the distance between F_1 and F_2 will not be the same before and after
                the transformation. For DC mobility feature F_2 will precede F_1 after
                the transformation.

Choice of the feature subject to this transformation is not restricted to the cutoff frequency of an original lowpass filter. In general it is possible to select any feature; e.g., the stopband edge, the DC, the deep minimum in the stopband, or other ones. The only condition is that the features must be selected in such a way that when creating N bands around the unit circle, there will be no band overlap.

This transformation can also be used for transforming other types of filters; e.g., notch filters or resonators can be easily replicated at a number of required frequency locations. A good application would be an adaptive tone cancellation circuit reacting to the changing number and location of tones.

Examples   Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
[z2,p2,k2] = zpklp2xn(z, p, k, [-0.5 0.5], [0 0.25], 'pass');
hfvt = fvtool(b, a, k2*poly(z2), poly(p2));
legend(hfvt,'Original Filter','Half-band Filter');
```

As demonstrated by the figure, the target filter has the desired response shape and values replicated from the prototype.

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter

Variable	Description
Wt	Desired frequency location in the transformed target filter
Pass	Choice ('pass'/'stop') of passband/stopband at DC, 'pass' being the default
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassDen	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between 0 and 1, with 1 corresponding to half the sample rate.

References   Cain, G.D., A. Krukowski and I. Kale, "High Order Transformations for Flexible IIR Filter Design," *VII European Signal Processing Conference (EUSIPCO'94)*, vol. 3, pp. 1582-1585, Edinburgh, United Kingdom, September 1994.

Krukowski, A., G.D. Cain and I. Kale, "Custom designed high-order frequency transformations for IIR filters," *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

See Also   zpkftransf | allpasslp2xn | iirlp2xn

zpkrateup

Purpose	Zero-pole-gain complex bandpass frequency transformation
Syntax	`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)`

Description

`[Z2,P2,K2,AllpassNum,AllpassDen] = zpkrateup(Z,P,K,N)` returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter being transformed from any prototype by applying an Nth-order rateup frequency transformation, where N is the upsample ratio. Transformation creates N equal replicas of the prototype filter frequency response.

It also returns the numerator, `AllpassNum`, and the denominator, `AllpassDen`, of the allpass mapping filter. The original lowpass filter is given with zeros, Z, poles, P, and gain factor, K.

Relative positions of other features of an original filter do not change in the target filter. This means that it is possible to select two features of an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still precede F_2 after the transformation. However, the distance between F_1 and F_2 will not be the same before and after the transformation.

Examples

Design a prototype real IIR halfband filter using a standard elliptic approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
% Upsample the prototype filter 4 times
[z2,p2,k2] = zpkrateup(z, p, k, 4);
% Compare prototype filter with target filter
fvtool(b, a, k2*poly(z2), poly(p2));
```

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter

Variable	Description
K	Gain factor of the prototype lowpass filter
N	Integer upsampling ratio
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassNum	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

See Also zpkrateup | allpassrateup | iirrateup

zpkshift

Purpose   Zero-pole-gain real shift frequency transformation

Syntax   [*Z2*,*P2*,*K2*,*AllpassNum*,*AllpassDen*] = zpkshift(*Z*,*P*,*K*,*Wo*,*Wt*)

Description   [*Z2*,*P2*,*K2*,*AllpassNum*,*AllpassDen*] = zpkshift(*Z*,*P*,*K*,*Wo*,*Wt*)
                 returns the zeros,*Z2* , poles, *P2*, and gain factor, *K2*, of the target filter
                 transformed from the zeros, poles, and gain factor of real lowpass
                 prototype by applying a second-order real shift frequency mapping.
                 It also returns the numerator, *AllpassNum*, and the denominator,
                 AllpassDen of the allpass mapping filter.

                 This transformation places one selected feature of an original filter,
                 located at frequency W_o, at the required target frequency location, W_t.
                 This transformation implements the "DC Mobility," which means that
                 the Nyquist feature stays at Nyquist, but the DC feature moves to a
                 location dependent on the selection of W_o and W_t.

                 Relative positions of other features of an original filter do not change in
                 the target filter. This means that it is possible to select two features of
                 an original filter, F_1 and F_2, with F_1 preceding F_2. Feature F_1 will still
                 precede F_2 after the transformation. However, the distance between F_1
                 and F_2 will not be the same before and after the transformation.

                 Choice of the feature subject to the real shift transformation is not
                 restricted to the cutoff frequency of an original lowpass filter. In general
                 it is possible to select any feature; e.g., the stopband edge, the DC, the
                 deep minimum in the stopband, or other ones.

                 This transformation can also be used for transforming other types of
                 filters; e.g., notch filters or resonators can change their position in a
                 simple way without the need to design them again.

Examples   Rotate frequency response by π/2 radians/sample:

```
[B,A] = ellip(10,0.1,40,0.25);
% Elliptic lowpass filter with passband frequency 0.25*pi rad/sample
Z = roots(B); % get roots of numerator polynomial- filter zeros
P = roots(A); % get roots of denominator polynomial- filter poles
K = B(1);
```

```
[Z2,P2,K2] = zpkshift(Z,P,K,0.25,0.75); % shift by 0.25*pi rad/sample
Num = poly(Z2);
Den = poly(P2);
hfvt = fvtool(B,A,K2*Num,Den);
legend(hfvt,'Original Filter','Rotate by \pi/2 radians/sample');
axis([0 1 -90 10]);
```


See Also   zpkftransf | allpassshift | iirshift

zpkshiftc

Purpose   Zero-pole-gain complex shift frequency transformation

Syntax   ```
[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,Wo,Wt)
[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,0.5)
[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,-0.5)
```
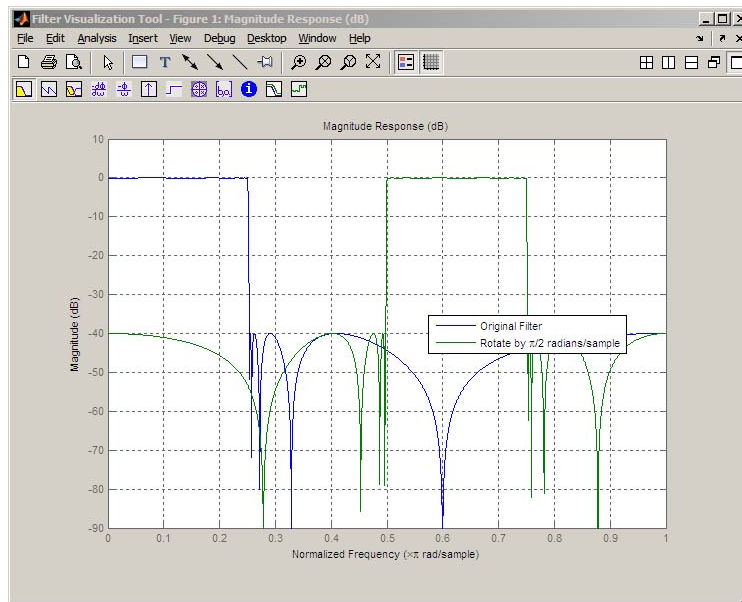
Description `[Z2,P2,K2,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,Wo,Wt)`
returns zeros, Z_2, poles, P_2, and gain factor, K_2, of the target filter
transformed from the real lowpass prototype by applying a first-order
complex frequency shift transformation. This transformation rotates all
the features of an original filter by the same amount specified by the
location of the selected feature of the prototype filter, originally at W_o,
placed at W_t in the target filter.

It also returns the numerator, `AllpassNum`, and the denominator,
`AllpassDen`, of the allpass mapping filter. The prototype lowpass filter
is given with zeros, `Z`, poles, `P`, and the gain factor, `K`.

`[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,0.5)`
performs the Hilbert transformation, i.e. a 90 degree counterclockwise
rotation of an original filter in the frequency domain.

`[Num,Den,AllpassNum,AllpassDen] = zpkshiftc(Z,P,K,0,-0.5)`
performs the inverse Hilbert transformation, i.e. a 90 degree clockwise
rotation of an original filter in the frequency domain.

Examples   Design a prototype real IIR halfband filter using a standard elliptic
approach:

```
[b, a] = ellip(3,0.1,30,0.409);
z = roots(b);
p = roots(a);
k = b(1);
```

Rotation by /4 Radians/Sample

Rotation by -0.25:

```
[z2,p2,k2] = zpkshiftc(z, p, k, 0.5, 0.25);
```

```
hfvt = fvtool(b,a,k2*poly(z2),poly(p2));
```

Rotation by /2 Radians/Sample

```
[z3,p3,k3] = zpkshiftc(z, p, k, 0, 0.5);
addfilter(hfvt,k3*poly(z3),poly(p3));
legend(hfvt,'Original Filter','Rotation by -\pi/4 radians/sample',...
    'Rotation by \pi/2 radians/sample');
```

Rotation by – /2 Radians/Sample

```
[z2,p2,k2] = zpkshiftc(z, p, k, 0.5, -0.5);
fvtool(b, a, k2*poly(z2), poly(p2));
```

Result of Example 1

After performing the rotation, the resulting filter shows the features desired.
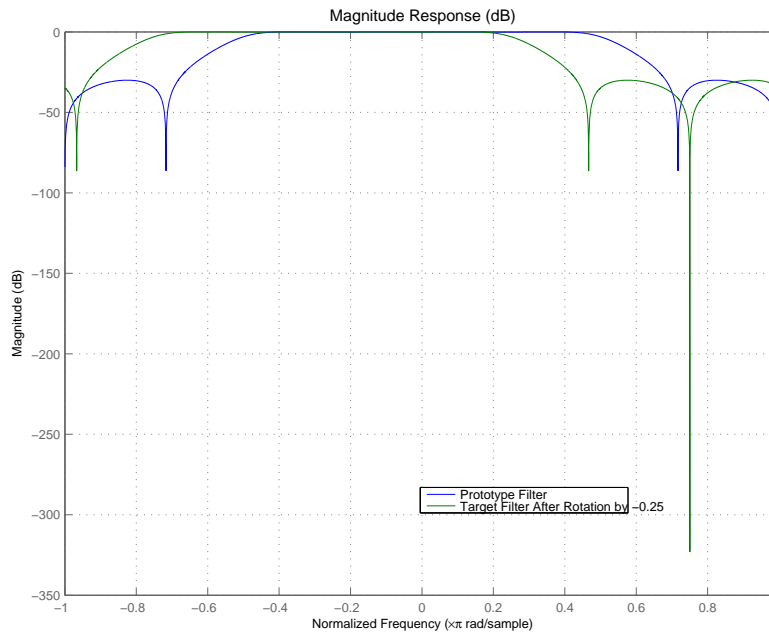
Result of Example 2

Similar to the first example, performing the Hilbert transformation generates the desired target filter, shown here.
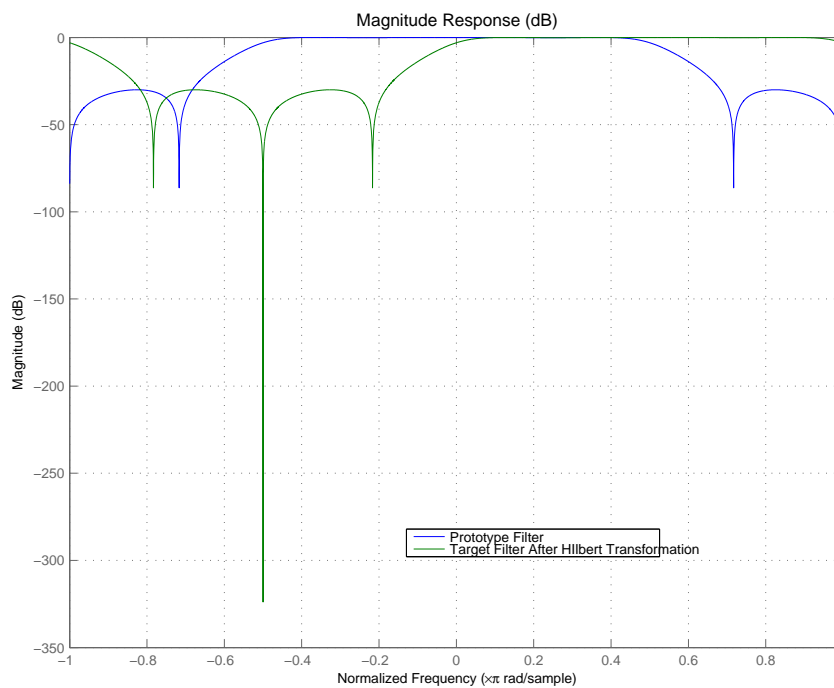
Result of Example 3

Finally, using the inverse Hilbert transformation creates yet a third filter, as the figure shows.

Arguments

Variable	Description
Z	Zeros of the prototype lowpass filter
P	Poles of the prototype lowpass filter
K	Gain factor of the prototype lowpass filter
Wo	Frequency value to be transformed from the prototype filter

zpkshiftc

Variable	Description
Wt	Desired frequency location in the transformed target filter
Z2	Zeros of the target filter
P2	Poles of the target filter
K2	Gain factor of the target filter
AllpassDen	Numerator of the mapping filter
AllpassDen	Denominator of the mapping filter

Frequencies must be normalized to be between -1 and 1, with 1 corresponding to half the sample rate.

References Oppenheim, A.V., R.W. Schafer and J.R. Buck, *Discrete-Time Signal Processing*, Prentice-Hall International Inc., 1989.

Dutta-Roy, S.C. and B. Kumar, "On digital differentiators, Hilbert transformers, and half-band low-pass filters," *IEEE Transactions on Education*, vol. 32, pp. 314-318, August 1989.

See Also zpkftransf | allpassshiftc | iirshiftc

Purpose   Zero-pole plot for filter

Syntax

```
zplane(Hq)
zplane(Hq,plotoption)
zplane(Hq,plotoption,plotoption2)
zplane(Hq,Name,Value)
[zq,pq,kq] = zplane(Hq)
[zq,pq,kq,zr,pr,kr] = zplane(Hq)
```

Description   zplane(Hq) plots the zeros and poles of a quantized filter Hq in the current figure window. The poles and zeros of the quantized and unquantized filters are plotted by default. The symbol o represents a zero of the unquantized reference filter, and the symbol x represents a pole of that filter. The symbols □ and + are used to plot the zeros and poles of the quantized filter Hq. The plot includes the unit circle for reference.

zplane(Hq,*plotoption*) plots the poles and zeros associated with the quantized filter Hq according to one specified plot option. The string *plotoption* can be either of the following reference filter display options:

- **'on'** to display the poles and zeros of both the quantized filter and the associated reference filter (default)

- **'off'** to display the poles and zeros of only the quantized filter

zplane(Hq,*plotoption*,*plotoption2*) plots the poles and zeros associated with the quantized filter Hq according to two specified plot options. The string *plotoption* can be selected from the reference filter display options listed in the previous syntax. The string *plotoption2* can be selected from the section-by-section plotting style options described in the following list:

- **'individual'** to display the poles and zeros of each section of the filter in a separate figure window

- **'overlay'** to display the poles and zeros of all sections of the filter on the same plot

zplane

- '**tile**' to display the poles and zeros of each section of the filter in a separate plot in the same figure window

zplane(Hq,Name,Value) returns a phase response with additional options specified by one or more Name,Value pair arguments. Name-value pair arguments are only available if Hq is a filter System object.

[zq,pq,kq] = zplane(Hq) returns the vectors of zeros zq, poles pq, and gains kq. If Hq has n sections, zq, pq, and kq are returned as 1-by-n cell arrays. If there are no zeros (or no poles), zq (or pq) is set to the empty matrix [].

[zq,pq,kq,zr,pr,kr] = zplane(Hq) returns the vectors of zeros zr, poles pr, and gains kr of the reference filter associated with the quantized filter Hq. It also returns the vectors of zeros zq, poles pq, and gains kq for the quantized filter Hq.

Input Arguments

Hq

hfilt is either:

- An adaptive adaptfilt, discrete-time dfilt, or multirate mfilt filter object

- A filter System object. The function supports the following filter structures

  - dsp.FIRFilter
  - dsp.BiquadFilter
  - dsp.FIRInterpolator
  - dsp.CICInterpolator
  - dsp.FIRDecimator
  - dsp.CICDecimator
  - dsp.FIRRateConverter

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Arithmetic
`double' | 'single' | 'fixed'

For filter System object inputs only, specify the arithmetic used during analysis. When you specify 'double' or 'single', the function performs double- or single-precision analysis. When you specify 'fixed' , the arithmetic changes depending on the setting of the CoefficientDataType property and whether the System object is locked or unlocked.

Details for Fixed-Point Arithmetic

System Object State	Coefficient Data Type	Rule
Unlocked	'Same as input'	The function assumes that the coefficient data type is signed, 16 bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Unlocked	'Custom'	The function performs fixed-point analysis based on the setting of the CustomCoefficientsDataType property.

System Object State	Coefficient Data Type	Rule
Locked	`'Same as input'`	When the input data type is `'double'` or `'fixed'`, the function assumes that the coefficient data type is signed, 16-bit, and autoscaled. The function performs fixed-point analysis based on this assumption.
Locked	`'Custom'`	The function performs fixed-point analysis based on the setting of the `CustomCoefficientsDataType` property.

When you do not specify the arithmetic for non-CIC structures, the function uses double-precision arithmetic if the filter System object is in an unlocked state. If the System object is locked, the function performs analysis based on the locked input data type. CIC structures only support fixed-point arithmetic.

Output Arguments

zq

Zeros of quantized filter.

pq

Poles of quantized filter.

kq

Gains of quantized filter.

zr

Zeros of reference filter.

pr

Poles of reference filter.

kr

Gains of reference filter.

Examples   Create a quantized filter Hq from a fourth-order digital filter with cutoff frequency of 0.6. Plot the quantized and unquantized poles and zeros associated with this quantized filter.

```
[b,a] = ellip(4,.5,20,.6);
Hq = dfilt.df2(b, a);
Hq.arithmetic = 'fixed';
zplane(Hq);
```
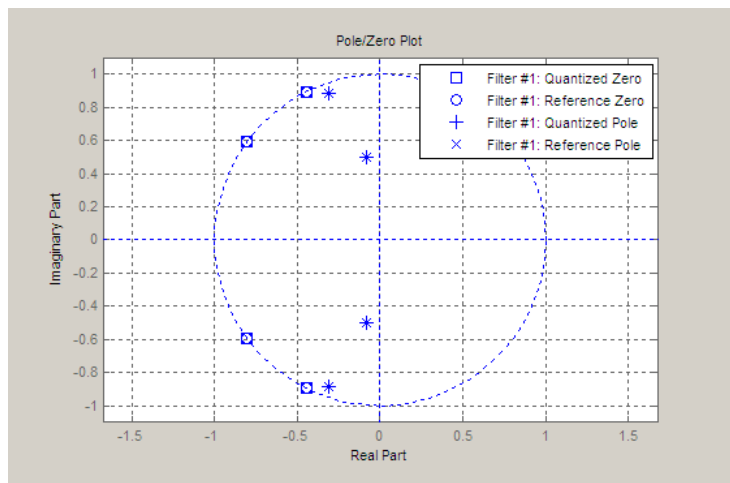

See Also   freqz | impz

zplane

8

Reference for the Properties of Filter Objects

Fixed-Point Filter Properties

In this section...

Overview of Fixed-Point Filters

There is a distinction between fixed-point filters and quantized filters — quantized filters represent a superset that includes fixed-point filters.

When `dfilt` objects have their `Arithmetic` property set to `single` or `fixed`, they are quantized filters. However, after you set the `Arithmetic` property to `fixed`, the resulting filter is both quantized and fixed-point. Fixed-point filters perform arithmetic operations without allowing the binary point to move in response to the calculation — hence the name fixed-point. You can find out more about fixed-point arithmetic in your Fixed-Point Toolbox documentation or from the Help system.

With the `Arithmetic` property set to `single`, meaning the filter uses single-precision floating-point arithmetic, the filter allows the binary point to move during mathematical operations, such as sums or products. Therefore these filters cannot be considered fixed-point filters. But they are quantized filters.

The following sections present the properties for fixed-point filters, which includes all the properties for double-precision and single-precision floating-point filters as well.

Fixed-Point Objects and Filters

Fixed-point filters depend in part on fixed-point objects from Fixed-Point Toolbox software. You can see this when you display a fixed-point filter at the command prompt.

```
hd=dfilt.df2t
```

```
hd =

         FilterStructure: 'Direct-Form II Transposed'
              Arithmetic: 'double'
                Numerator: 1
              Denominator: 1
         PersistentMemory: false
                   States: [0x1 double]

set(hd,'arithmetic','fixed')
hd

hd =

         FilterStructure: 'Direct-Form II Transposed'
              Arithmetic: 'fixed'
                Numerator: 1
              Denominator: 1
         PersistentMemory: false
                   States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                   Signed: true

         InputWordLength: 16
          InputFracLength: 15

        OutputWordLength: 16
         OutputFracLength: 15

         StateWordLength: 16
          StateAutoScale: true

              ProductMode: 'FullPrecision'

         AccumWordLength: 40
            CastBeforeSum: true
```

```
                    RoundMode: 'convergent'
                 OverflowMode: 'wrap'
```

Look at the States property, shown here

```
States: [1x1 embedded.fi]
```

The notation `embedded.fi` indicates that the states are being represented by fixed-point objects, usually called `fi` objects. If you take a closer look at the property `States`, you see how the properties of the `fi` object represent the values for the filter states.

```
hd.states

ans =

[]

                 DataType: Fixed
                  Scaling: BinaryPoint
                   Signed: true
               WordLength: 16
           FractionLength: 15

                RoundMode: round
             OverflowMode: saturate
              ProductMode: FullPrecision
      MaxProductWordLength: 128
                  SumMode: FullPrecision
          MaxSumWordLength: 128
             CastBeforeSum: true
```

To learn more about `fi` objects (fixed-point objects) in general, refer to your Fixed-Point Toolbox documentation.

As inputs (data to be filtered), fixed-point filters accept both regular double-precision values and `fi` objects. Which you use depends on your needs. How your filter responds to the input data is determined by the settings of the filter properties, discussed in the next few sections.

Summary — Fixed-Point Filter Properties

Discrete-time filters in this toolbox use objects that perform the filtering and configuration of the filter. As objects, they include properties and methods that are often referred to as functions — not strictly the same as MATLAB functions but mostly so) to provide filtering capability. In discrete-time filters, or `dfilt` objects, many of the properties are dynamic, meaning they become available depending on the settings of other properties in the `dfilt` object or filter.

Dynamic Properties

When you use a `dfilt.`*`structure`* function to create a filter, MATLAB displays the filter properties in the command window in return (unless you end the command with a semicolon which suppresses the output display). Generally you see six or seven properties, ranging from the property `FilterStructure` to `PersistentMemory`. These first properties are always present in the filter. One of the most important properties is `Arithmetic`. The `Arithmetic` property controls all of the dynamic properties for a filter.

Dynamic properties become available when you change another property in the filter. For example, when you change the `Arithmetic` property value to `fixed`, the display now shows many more properties for the filter, all of them considered dynamic. Here is an example that uses a direct form II filter. First create the default filter:

```
hd=dfilt.df2

hd =

        FilterStructure: 'Direct-Form II'
             Arithmetic: 'double'
               Numerator: 1
             Denominator: 1
        PersistentMemory: false
                  States: [0x1 double]
```

With the filter `hd` in the workspace, convert the arithmetic to fixed-point. Do this by setting the property `Arithmetic` to `fixed`. Notice the display. Instead of a few properties, the filter now has many more, each one related

to a particular part of the filter and its operation. Each of the now-visible properties is dynamic.

```
hd.arithmetic='fixed'

hd =

        FilterStructure: 'Direct-Form II'
              Arithmetic: 'fixed'
               Numerator: 1
             Denominator: 1
        PersistentMemory: false
                  States: [1x1 embedded.fi]

        CoeffWordLength: 16
         CoeffAutoScale: true
                 Signed: true

        InputWordLength: 16
        InputFracLength: 15

       OutputWordLength: 16
             OutputMode: 'AvoidOverflow'

        StateWordLength: 16
        StateFracLength: 15

            ProductMode: 'FullPrecision'

        AccumWordLength: 40
          CastBeforeSum: true

              RoundMode: 'convergent'
           OverflowMode: 'wrap'
```

Even this list of properties is not yet complete. Changing the value of other properties such as the ProductMode or CoeffAutoScale properties may reveal even more properties that control how the filter works. Remember this feature about dfilt objects and dynamic properties as you review the rest of this section about properties of fixed-point filters.

An important distinction is you cannot change the value of a property unless you see the property listed in the default display for the filter. Entering the filter name at the MATLAB prompt generates the default property display for the named filter. Using get(filtername) does not generate the default display — it lists all of the filter properties, both those that you can change and those that are not available yet.

The following table summarizes the properties, static and dynamic, of fixed-point filters and provides a brief description of each. Full descriptions of each property, in alphabetical order, follow the table.

Property Name	Valid Values [Default Value]	Brief Description
AccumFracLength	Any positive or negative integer number of bits [29]	Specifies the fraction length used to interpret data output by the accumulator. This is a property of FIR filters and lattice filters. IIR filters have two similar properties — DenAccumFracLength and NumAccumFracLength — that let you set the precision for numerator and denominator operations separately.
AccumWordLength	Any positive integer number of bits [40]	Sets the word length used to store data in the accumulator/buffer.
Arithmetic	[Double], single, fixed	Defines the arithmetic the filter uses. Gives you the options double, single, and fixed. In short, this property defines the operating mode for your filter.
CastBeforeSum	[True] or false	Specifies whether to cast numeric data to the appropriate accumulator format (as shown in the signal flow diagrams) before performing sum operations.

Property Name	Valid Values [Default Value]	Brief Description
CoeffAutoScale	[True] or false	Specifies whether the filter automatically chooses the proper fraction length to represent filter coefficients without overflowing. Turning this off by setting the value to false enables you to change the NumFracLength and DenFracLength properties to specify the precision used.
CoeffFracLength	Any positive or negative integer number of bits [14]	Set the fraction length the filter uses to interpret coefficients. CoeffFracLength is not available until you set CoeffAutoScale to false. Scalar filters include this property.
CoeffWordLength	Any positive integer number of bits [16]	Specifies the word length to apply to filter coefficients.
DenAccumFracLength	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of addition operations involving denominator coefficients.
DenFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length the filter uses to interpret denominator coefficients. DenFracLength is always available, but it is read-only until you set CoeffAutoScale to false.
Denominator	Any filter coefficient value [1]	Holds the denominator coefficients for IIR filters.
DenProdFracLength	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of product operations involving denominator coefficients. You can change this property value after you set ProductMode to SpecifyPrecision.

Property Name	Valid Values [Default Value]	Brief Description
DenStateFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length used to interpret the states associated with denominator coefficients in the filter.
FracDelay	Any decimal value between 0 and 1 samples	Specifies the fractional delay provided by the filter, in decimal fractions of a sample.
FDAutoScale	[True] or false	Specifies whether the filter automatically chooses the proper scaling to represent the fractional delay value without overflowing. Turning this off by setting the value to false enables you to change the FDWordLength and FDFracLength properties to specify the data format applied.
FDFracLength	Any positive or negative integer number of bits [5]	Specifies the fraction length to represent the fractional delay.
FDProdFracLength	Any positive or negative integer number of bits [34]	Specifies the fraction length to represent the result of multiplying the coefficients with the fractional delay.
FDProdWordLength	Any positive or negative integer number of bits [39]	Specifies the word length to represent result of multiplying the coefficients with the fractional delay.
FDWordLength	Any positive or negative integer number of bits [6]	Specifies the word length to represent the fractional delay.
DenStateWordLength	Any positive integer number of bits [16]	Specifies the word length used to represent the states associated with denominator coefficients in the filter.

Property Name	Valid Values [Default Value]	Brief Description
FilterInternals	[FullPrecision], SpecifyPrecision	Controls whether the filter sets the output word and fraction lengths, and the accumulator word and fraction lengths automatically to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.
FilterStructure	Not applicable.	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output.
InputFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length the filter uses to interpret data to be processed by the filter.
InputWordLength	Any positive integer number of bits [16]	Specifies the word length applied to represent input data.
Ladder	Any ladder coefficients in double-precision data type [1]	latticearma filters include this property to store the ladder coefficients.
LadderAccumFrac Length	Any positive or negative integer number of bits [29]	latticearma filters use this to define the fraction length applied to values output by the accumulator that stores the results of ladder computations.
LadderFracLength	Any positive or negative integer number of bits [14]	latticearma filters use ladder coefficients in the signal flow. This property determines the fraction length used to interpret the coefficients.

Property Name	Valid Values [Default Value]	Brief Description
Lattice	Any lattice structure coefficients. No default value.	Stores the lattice coefficients for lattice-based filters.
LatticeAccumFrac Length	Any positive or negative integer number of bits [29]	Specifies how the accumulator outputs the results of operations on the lattice coefficients.
LatticeFracLength	Any positive or negative integer number of bits [15]	Specifies the fraction length applied to the lattice coefficients.
MultiplicandFrac Length	Any positive or negative integer number of bits [15]	Sets the fraction length for values used in product operations in the filter. Direct-form I transposed (df1t) filter structures include this property.
MultiplicandWord Length	Any positive integer number of bits [16]	Sets the word length applied to the values input to a multiply operation (the multiplicands). The filter structure df1t includes this property.
NumAccumFracLength	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of addition operations involving numerator coefficients.
Numerator	Any double-precision filter coefficients [1]	Holds the numerator coefficient values for the filter.
NumFracLength	Any positive or negative integer number of bits [14]	Sets the fraction length used to interpret the numerator coefficients.
NumProdFracLength	Any positive or negative integer number of bits [29]	Specifies how the filter algorithm interprets the results of product operations involving numerator coefficients. You can change the property value after you set ProductMode to SpecifyPrecision.

Property Name	Valid Values [Default Value]	Brief Description
NumStateFracLength	Any positive or negative integer number of bits [15]	For IIR filters, this defines the fraction length applied to the numerator states of the filter. Specifies the fraction length used to interpret the states associated with numerator coefficients in the filter.
NumStateWordLength	Any positive integer number of bits [16]	For IIR filters, this defines the word length applied to the numerator states of the filter. Specifies the word length used to interpret the states associated with numerator coefficients in the filter.
OutputFracLength	Any positive or negative integer number of bits — [15] or [12] bits depending on the filter structure	Determines how the filter interprets the filtered data. You can change the value of OutputFracLength after you set OutputMode to SpecifyPrecision.
OutputMode	[AvoidOverflow], BestPrecision, SpecifyPrecision	Sets the mode the filter uses to scale the filtered input data. You have the following choices: • AvoidOverflow — directs the filter to set the output data fraction length to avoid causing the data to overflow. • BestPrecision — directs the filter to set the output data fraction length to maximize the precision in the output data. • SpecifyPrecision — lets you set the fraction length used by the filtered data.
OutputWordLength	Any positive integer number of bits [16]	Determines the word length used for the filtered data.

Property Name	Valid Values [Default Value]	Brief Description
OverflowMode	Saturate or [wrap]	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic. The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — hey maintain full precision.
ProductFracLength	Any positive or negative integer number of bits [29]	For the output from a product operation, this sets the fraction length used to interpret the numeric data. This property becomes writable (you can change the value) after you set ProductMode to SpecifyPrecision.
ProductMode	[FullPrecision], KeepLSB, KeepMSB, SpecifyPrecision	Determines how the filter handles the output of product operations. Choose from full precision (FullPrecision), or whether to keep the most significant bit (KeepMSB) or least significant bit (KeepLSB) in the result when you need to shorten the data words. For you to be able to set the precision (the fraction length) used by the output from the multiplies, you set ProductMode to SpecifyPrecision.

Property Name	Valid Values [Default Value]	Brief Description
ProductWordLength	Any positive number of bits. Default is 16 or 32 depending on the filter structure	Specifies the word length to use for the results of multiplication operations. This property becomes writable (you can change the value) after you set ProductMode to SpecifyPrecision.
PersistentMemory	True or [false]	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states from previous filtering runs. True is the default setting.
RoundMode	[Convergent], ceil, fix, floor, nearest, round	Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software. • fix - Round toward zero. • floor - Round toward negative infinity. • nearest - Round toward nearest. Ties round toward positive infinity. • round - Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.

Property Name	Valid Values [Default Value]	Brief Description
		The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.
ScaleValueFracLength	Any positive or negative integer number of bits [29]	Scale values work with SOS filters. Setting this property controls how your filter interprets the scale values by setting the fraction length. Available only when you disable CoeffAutoScale by setting it to false.
ScaleValues	[2 x 1 double] array with values of 1	Stores the scaling values for sections in SOS filters.
Signed	[True] or false	Specifies whether the filter uses signed or unsigned fixed-point coefficients. Only coefficients reflect this property setting.
sosMatrix	[1 0 0 1 0 0]	Holds the filter coefficients as property values. Displays the matrix in the format [sections x coefficients/section datatype]. A [15x6 double] SOS matrix represents a filter with 6 coefficients per section and 15 sections, using data type double to represent the coefficients.

Property Name	Valid Values [Default Value]	Brief Description
SectionInputAuto Scale	[True] or false	Specifies whether the filter automatically chooses the proper fraction length to prevent overflow by data entering a section of an SOS filter. Setting this property to `false` enables you to change the `SectionInputFracLength` property to specify the precision used. Available only for SOS filters.
SectionInputFrac Length	Any positive or negative integer number of bits [29]	Section values work with SOS filters. Setting this property controls how your filter interprets the section values between sections of the filter by setting the fraction length. This applies to data entering a section. Compare to SectionOutputFracLength. Available only when you disable SectionInputAutoScale by setting it to `false`.
SectionInputWord Length	Any positive or negative integer number of bits [29]	Sets the word length used to represent the data moving into a section of an SOS filter.
SectionOutputAuto Scale	[True] or false	Specifies whether the filter automatically chooses the proper fraction length to prevent overflow by data leaving a section of an SOS filter. Setting this property to `false` enables you to change the SectionOutputFracLength property to specify the precision used.

Property Name	Valid Values [Default Value]	Brief Description
`SectionOutputFrac Length`	Any positive or negative integer number of bits [29]	Section values work with SOS filters. Setting this property controls how your filter interprets the section values between sections of the filter by setting the fraction length. This applies to data leaving a section. Compare to `SectionInputFracLength`. Available after you disable `SectionOutputAutoScale` by setting it to `false`.
`SectionOutputWord Length`	Any positive or negative integer number of bits [32]	Sets the word length used to represent the data moving out of one section of an SOS filter.
`StateFracLength`	Any positive or negative integer number of bits [15]	Lets you set the fraction length applied to interpret the filter states.
`States`	[1x1 embedded `fi`]	Contains the filter states before, during, and after filter operations. States act as filter memory between filtering runs or sessions. Notice that the states use `fi` objects, with the associated properties from those objects. For details, refer to `filtstates` in your Signal Processing Toolbox documentation or in the Help system.
`StateWordLength`	Any positive integer number of bits [16]	Sets the word length used to represent the filter states.
`TapSumFracLength`	Any positive or negative integer number of bits [15]	Sets the fraction length used to represent the filter tap values in addition operations. This is available after you set `TapSumMode` to `false`. Symmetric and antisymmetric FIR filters include this property.

Property Name	Valid Values [Default Value]	Brief Description
TapSumMode	FullPrecision, KeepLSB, [KeepMSB], SpecifyPrecision	Determines how the accumulator outputs stored that involve filter tap weights. Choose from full precision (FullPrecision) to prevent overflows, or whether to keep the most significant bits (KeepMSB) or least significant bits (KeepLSB) when outputting results from the accumulator. To let you set the precision (the fraction length) used by the output from the accumulator, set FilterInternals to SpecifyPrecision. Symmetric and antisymmetric FIR filters include this property.
TapSumWordLength	Any positive number of bits [17]	Sets the word length used to represent the filter tap weights during addition. Symmetric and antisymmetric FIR filters include this property.

Property Details for Fixed-Point Filters

When you create a fixed-point filter, you are creating a filter object (a dfilt object). In this manual, the terms filter, dfilt object, and filter object are used interchangeably. To filter data, you apply the filter object to your data set. The output of the operation is the data filtered by the filter and the filter property values.

Filter objects have properties to which you assign property values. You use these property values to assign various characteristics to the filters you create, including

- The type of arithmetic to use in filtering operations

- The structure of the filter used to implement the filter (not a property you can set or change — you select it by the dfilt.*structure* function you choose)

- The locations of quantizations and cast operations in the filter

- The data formats used in quantizing, casting, and filtering operations

Details of the properties associated with fixed-point filters are described in alphabetical order on the following pages.

AccumFracLength

Except for state-space filters, all `dfilt` objects that use fixed arithmetic have this property that defines the fraction length applied to data in the accumulator. Combined with `AccumWordLength`, `AccumFracLength` helps fully specify how the accumulator outputs data after processing addition operations. As with all fraction length properties, `AccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers.

AccumWordLength

You use `AccumWordLength` to define the data word length used in the accumulator. Set this property to a value that matches your intended hardware. For example, many digital signal processors use 40-bit accumulators, so set `AccumWordLength` to 40 in your fixed-point filter:

```
set(hq,'arithmetic','fixed');
set(hq,'AccumWordLength',40);
```

Note that `AccumWordLength` only applies to filters whose `Arithmetic` property value is `fixed`.

Arithmetic

Perhaps the most important property when you are working with `dfilt` objects, `Arithmetic` determines the type of arithmetic the filter uses, and the properties or quantizers that compose the fixed-point or quantized filter. You use strings to set the `Arithmetic` property value.

The next table shows the valid strings for the Arithmetic property. Following the table, each property string appears with more detailed information about what happens when you select the string as the value for `Arithmetic` in your `dfilt`.

Arithmetic Property String	Brief Description of Effect on the Filter
double	All filtering operations and coefficients use double-precision floating-point representations and math. When you use dfilt.*structure* to create a filter object, double is the default value for the Arithmetic property.
single	All filtering operations and coefficients use single-precision floating-point representations and math.
fixed	This string applies selected default values for the properties in the fixed-point filter object, including such properties as coefficient word lengths, fraction lengths, and various operating modes. Generally, the default values match those you use on many digital signal processors. Allows signed fixed data types only. Fixed-point arithmetic filters are available only when you install Fixed-Point Toolbox software with this toolbox.

double. When you use one of the dfilt.*structure* methods to create a filter, the Arithmetic property value is double by default. Your filter is identical to the same filter without the Arithmetic property, as you would create if you used Signal Processing Toolbox software.

Double means that the filter uses double-precision floating-point arithmetic in all operations while filtering:

- All input to the filter must be double data type. Any other data type returns an error.

- The states and output are doubles as well.

- All internal calculations are done in double math.

When you use double data type filter coefficients, the reference and quantized (fixed-point) filter coefficients are identical. The filter stores the reference coefficients as double data type.

single. When your filter should use single-precision floating-point arithmetic, set the `Arithmetic` property to `single` so all arithmetic in the filter processing gets restricted to single-precision data type.

- Input data must be single data type. Other data types return errors.
- The filter states and filter output use single data type.

When you choose `single`, you can provide the filter coefficients in either of two ways:

- Double data type coefficients. With `Arithmetic` set to `single`, the filter casts the double data type coefficients to single data type representation.
- Single data type. These remain unchanged by the filter.

Depending on whether you specified single or double data type coefficients, the reference coefficients for the filter are stored in the data type you provided. If you provide coefficients in double data type, the reference coefficients are double as well. Providing single data type coefficients generates single data type reference coefficients. Note that the arithmetic used by the reference filter is always double.

When you use `reffilter` to create a reference filter from the reference coefficients, the resulting filter uses double-precision versions of the reference filter coefficients.

To set the `Arithmetic` property value, create your filter, then use `set` to change the `Arithmetic` setting, as shown in this example using a direct form FIR filter.

```
b=fir1(7,0.45);

hd=dfilt.dffir(b)

hd =

         FilterStructure: 'Direct-Form FIR'
               Arithmetic: 'double'
                 Numerator: [1x8 double]
          PersistentMemory: false
```

```
                     States: [7x1 double]

set(hd,'arithmetic','single')
hd

hd =

        FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'single'
               Numerator: [1x8 double]
         PersistentMemory: false
                   States: [7x1 single]
```

fixed. Converting your `dfilt` object to use fixed arithmetic results in a filter structure that uses properties and property values to match how the filter would behave on digital signal processing hardware.

Note The `fixed` option for the property `Arithmetic` is available only when you install Fixed-Point Toolbox software as well as DSP System Toolbox software.

After you set `Arithmetic` to `fixed`, you are free to change any property value from the default value to a value that more closely matches your needs. You cannot, however, mix floating-point and fixed-point arithmetic in your filter when you select `fixed` as the `Arithmetic` property value. Choosing `fixed` restricts you to using either fixed-point or floating point throughout the filter (the data type must be homogenous). Also, all data types must be signed. `fixed` does not support unsigned data types except for unsigned coefficients when you set the property `Signed` to `false`. Mixing word and fraction lengths within the fixed object is acceptable. In short, using fixed arithmetic assumes

- fixed word length.

- fixed size and dedicated accumulator and product registers.

- the ability to do either saturation or wrap arithmetic.

- that multiple rounding modes are available.

Making these assumptions simplifies your job of creating fixed-point filters by reducing repetition in the filter construction process, such as only requiring you to enter the accumulator word size once, rather than for each step that uses the accumulator.

Default property values are a starting point in tailoring your filter to common hardware, such as choosing 40-bit word length for the accumulator, or 16-bit words for data and coefficients.

In this `dfilt` object example, `get` returns the default values for `dfilt.df1t` structures.

```
[b,a]=butter(6,0.45);
hd=dfilt.df1(b,a)

hd =

          FilterStructure: 'Direct-Form I'
                Arithmetic: 'double'
                 Numerator: [1x7 double]
               Denominator: [1x7 double]
          PersistentMemory: false
                    States: Numerator:  [6x1 double]
                            Denominator:[6x1 double]


set(hd,'arithmetic','fixed')
get(hd)
        PersistentMemory: false
         FilterStructure: 'Direct-Form I'
                  States: [1x1 filtstates.dfiir]
               Numerator: [1x7 double]
             Denominator: [1x7 double]
              Arithmetic: 'fixed'
          CoeffWordLength: 16
           CoeffAutoScale: 1
                   Signed: 1
                RoundMode: 'convergent'
             OverflowMode: 'wrap'
          InputWordLength: 16
```

```
            InputFracLength: 15
                ProductMode: 'FullPrecision'
          OutputWordLength: 16
          OutputFracLength: 15
             NumFracLength: 16
             DenFracLength: 14
         ProductWordLength: 32
        NumProdFracLength: 31
        DenProdFracLength: 29
           AccumWordLength: 40
       NumAccumFracLength: 31
       DenAccumFracLength: 29
             CastBeforeSum: 1
```

Here is the default display for hd.

hd

hd =

```
            FilterStructure: 'Direct-Form I'
                 Arithmetic: 'fixed'
                  Numerator: [1x7 double]
                Denominator: [1x7 double]
           PersistentMemory: false
                     States: Numerator:  [6x1 fi]
                             Denominator:[6x1 fi]


           CoeffWordLength: 16
            CoeffAutoScale: true
                    Signed: true

           InputWordLength: 16
           InputFracLength: 15

          OutputWordLength: 16
          OutputFracLength: 15

                ProductMode: 'FullPrecision'
```

```
         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

This second example shows the default property values for
dfilt.latticemamax filter objects, using the coefficients from an fir1 filter.

```
b=fir1(7,0.45)


hdlat=dfilt.latticemamax(b)

hdlat =

          FilterStructure: [1x45 char]
                Arithmetic: 'double'
                    Lattice: [1x8 double]
         PersistentMemory: false
                    States: [8x1 double]

hdlat.arithmetic='fixed'

hdlat =

          FilterStructure: [1x45 char]
                Arithmetic: 'fixed'
                    Lattice: [1x8 double]
         PersistentMemory: false
                    States: [1x1 embedded.fi]

          CoeffWordLength: 16
           CoeffAutoScale: true
                    Signed: true

           InputWordLength: 16
           InputFracLength: 15
```

```
        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

         StateWordLength: 16
         StateFracLength: 15

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'
```

Unlike the `single` or `double` options for `Arithmetic`, `fixed` uses properties to define the word and fraction lengths for each portion of your filter. By changing the property value of any of the properties, you control your filter performance. Every word length and fraction length property is independent — set the one you need and the others remain unchanged, such as setting the input word length with `InputWordLength`, while leaving the fraction length the same.

```
d=fdesign.lowpass('n,fc',6,0.45)

d =

               Response: 'Lowpass with cutoff'
          Specification: 'N,Fc'
            Description: {2x1 cell}
    NormalizedFrequency: true
                     Fs: 'Normalized'
             FilterOrder: 6
                 Fcutoff: 0.4500


designmethods(d)


Design Methods for class fdesign.lowpass:
```

```
butter

hd=butter(d)

hd =

 FilterStructure: 'Direct-Form II, Second-Order Sections'
      Arithmetic: 'double'
        sosMatrix: [3x6 double]
     ScaleValues: [4x1 double]
PersistentMemory: false
          States: [2x3 double]

hd.arithmetic='fixed'

hd =

  FilterStructure: 'Direct-Form II, Second-Order Sections'
       Arithmetic: 'fixed'
         sosMatrix: [3x6 double]
      ScaleValues: [4x1 double]
 PersistentMemory: false
           States: [1x1 embedded.fi]

  CoeffWordLength: 16
   CoeffAutoScale: true
           Signed: true

  InputWordLength: 16
  InputFracLength: 15

 SectionInputWordLength: 16
  SectionInputAutoScale: true

 SectionOutputWordLength: 16
 Section OutputAutoScale: true

       OutputWordLength: 16
              OutputMode: 'AvoidOverflow'
```

```
            StateWordLength: 16
            StateFracLength: 15

                ProductMode: 'FullPrecision'

            AccumWordLength: 40
              CastBeforeSum: true

                  RoundMode: 'convergent'
               OverflowMode: 'wrap'

hd.inputWordLength=12

hd =

 FilterStructure: 'Direct-Form II, Second-Order Sections'
       Arithmetic: 'fixed'
         sosMatrix: [3x6 double]
       ScaleValues: [4x1 double]
PersistentMemory: false
                    States: [1x1 embedded.fi]

   CoeffWordLength: 16
    CoeffAutoScale: true
            Signed: true

   InputWordLength: 12
   InputFracLength: 15

   SectionInputWordLength: 16
    SectionInputAutoScale: true

 SectionOutputWordLength: 16
  SectionOutputAutoScale: true

         OutputWordLength: 16
               OutputMode: 'AvoidOverflow'

          StateWordLength: 16
```

```
        StateFracLength: 15

            ProductMode: 'FullPrecision'

       AccumWordLength: 40
          CastBeforeSum: true

              RoundMode: 'convergent'
           OverflowMode: 'wrap'
```

Notice that the properties for the lattice filter `hdlat` and direct-form II filter `hd` are different, as befits their differing filter structures. Also, some properties are common to both objects, such as `RoundMode` and `PersistentMemory` and behave the same way in both objects.

Notes About Fraction Length, Word Length, and Precision. Word length and fraction length combine to make the format for a fixed-point number, where word length is the number of bits used to represent the value and fraction length specifies, in bits, the location of the binary point in the fixed-point representation. Therein lies a problem — fraction length, which you specify in bits, can be larger than the word length, or a negative number of bits. This section explains how that idea works and how you might use it.

Unfortunately fraction length is somewhat misnamed (although it continues to be used in this User's Guide and elsewhere for historical reasons).

Fraction length defined as the number of fractional bits (bits to the right of the binary point) is true only when the fraction length is positive and less than or equal to the word length. In MATLAB format notation you can use [word length fraction length]. For example, for the format [16 16], the second 16 (the fraction length) is the number of fractional bits or bits to the right of the binary point. In this example, all 16 bits are to the right of the binary point.

But it is also possible to have fixed-point formats of [16 18] or [16 -45]. In these cases the fraction length can no longer be the number of bits to the right of the binary point since the format says the word length is 16 — there cannot be 18 fraction length bits on the right. And how can there be a negative number of bits for the fraction length, such as [16 -45]?

A better way to think about fixed-point format [word length fraction length] and what it means is that the representation of a fixed-point number is a weighted sum of powers of two driven by the fraction length, or the two's complement representation of the fixed-point number.

Consider the format [B L], where the fraction length L can be positive, negative, 0, greater than B (the word length) or less than B. (B and L are always integers and B is always positive.)

Given a binary string b(1) b(2) b(3) ... b(B), to determine the two's-complement value of the string in the format described by [B L], use the value of the individual bits in the binary string in the following formula, where b(1) is the first binary bit (and most significant bit, MSB), b(2) is the second, and on up to b(B).

The decimal numeric value that those bits represent is given by

```
value =-b(1)*2^(B-L-1)+b(2)*2^(B-L-2)+b(3)*2^(B-L-3)+...+ b(B)*2^(-L)
```

L, the fraction length, represents the negative of the weight of the last, or least significant bit (LSB). L is also the step size or the precision provided by a given fraction length.

Precision. Here is how precision works.

When all of the bits of a binary string are zero except for the LSB (which is therefore equal to one), the value represented by the bit string is given by $2^{(-L)}$. If L is negative, for example L=-16, the value is 2^{16}. The smallest step between numbers that can be represented in a format where L=-16 is given by 1×2^{16} (the rightmost term in the formula above), which is 65536. Note the precision does not depend on the word length.

Take a look at another example. When the word length set to 8 bits, the decimal value 12 is represented in binary by 00001100. That 12 is the decimal equivalent of 00001100 tells you that you are using [8 0] data format representation — the word length is 8 bits and fraction length 0 bits, and the step size or precision (the smallest difference between two adjacent values in the format [8,0], is 2^0=1.

Suppose you plan to keep only the upper 5 bits and discard the other three. The resulting precision after removing the right-most three bits comes from the weight of the lowest remaining bit, the fifth bit from the left, which is $2^3=8$, so the format would be [5,-3].

Note that in this format the step size is 8, I cannot represent numbers that are between multiples of 8.

In MATLAB, with Fixed-Point Toolbox software installed:

```
x=8;
q=quantizer([8,0]); % Word length = 8, fraction length = 0
xq=quantize(q,x);
binxq=num2bin(q,xq);
q1=quantizer([5 -3]); % Word length = 5, fraction length = -3
xq1 = quantize(q1,xq);
binxq1=num2bin(q1,xq1);
binxq

binxq =

00001000

binxq1

binxq1 =

00001
```

But notice that in [5,-3] format, 00001 is the two's complement representation for 8, not for 1; q = quantizer([8 0]) and q1 = quantizer([5 -3]) are not the same. They cover the about the same range — range(q)>range(q1) — but their quantization step is different — eps(q)= 8, and eps(q1)=1.

Look at one more example. When you construct a quantizer q

```
q = quantizer([a,b])
```

the first element in [a,b] is a, the word length used for quantization. The second element in the expression, b, is related to the quantization step — the numerical difference between the two closest values that the quantizer

can represent. This is also related to the weight given to the LSB. Note that `2^(-b) = eps(q)`.

Now construct two quantizers, `q1` and `q2`. Let `q1` use the format [32,0] and let `q2` use the format [16, -16].

```
q1 = quantizer([32,0])
q2 = quantizer([16,-16])
```

Quantizers `q1` and `q2` cover the same range, but `q2` has less precision. It covers the range in steps of 2^{16}, while `q` covers the range in steps of 1.

This lost precision is due to (or can be used to model) throwing out 16 least-significant bits.

An important point to understand is that in `dfilt` objects and filtering you control which bits are carried from the sum and product operations in the filter to the filter output by setting the format for the output from the sum or product operation.

For instance, if you use [16 0] as the output format for a 32-bit result from a sum operation when the original format is [32 0], you take the lower 16 bits from the result. If you use [16 -16], you take the higher 16 bits of the original 32 bits. You could even take 16 bits somewhere in between the 32 bits by choosing something like [16 -8], but you probably do not want to do that.

Filter scaling is directly implicated in the format and precision for a filter. When you know the filter input and output formats, as well as the filter internal formats, you can scale the inputs or outputs to stay within the format ranges. For more information about scaling filters, refer to "Floating-Point to Fixed-Point Filter Conversion".

Notice that overflows or saturation might occur at the filter input, filter output, or within the filter itself, such as during add or multiply or accumulate operations. Improper scaling at any point in the filter can result in numerical errors that dramatically change the performance of your fixed-point filter implementation.

CastBeforeSum

Setting the `CastBeforeSum` property determines how the filter handles the input values to sum operations in the filter. After you set your filter `Arithmetic` property value to `fixed`, you have the option of using `CastBeforeSum` to control the data type of some inputs (addends) to summations in your filter. To determine which addends reflect the `CastBeforeSum` property setting, refer to the reference page for the signal flow diagram for the filter structure.

`CastBeforeSum` specifies whether to cast selected addends to summations in the filter to the output format from the addition operation before performing the addition. When you specify `true` for the property value, the results of the affected sum operations match most closely the results found on most digital signal processors. Performing the cast operation before the summation adds one or two additional quantization operations that can add error sources to your filter results.

Specifying `CastBeforeSum` to be `false` prevents the addends from being cast to the output format before the addition operation. Choose this setting to get the most accurate results from summations without considering the hardware your filter might use.

Notice that the output format for every sum operation reflects the value of the output property specified in the filter structure diagram. Which input property is referenced by `CastBeforeSum` depends on the structure.

Property Value	Description
false	Configures filter summation operations to retain the addends in the format carried from the previous operation.
true	Configures filter summation operations to convert the input format of the addends to match the summation output format before performing the summation operation. Usually this generates results from the summation that more closely match those found from digital signal processors

Another point — with `CastBeforeSum` set to `false`, the filter realization process inserts an intermediate data type format to hold temporarily the full precision sum of the inputs. A separate Convert block performs the process of casting the addition result to the accumulator format. This intermediate data format occurs because the Sum block in Simulink always casts input (addends) to the output data type.

Diagrams of CastBeforeSum Settings. When `CastBeforeSum` is `false`, sum elements in filter signal flow diagrams look like this:
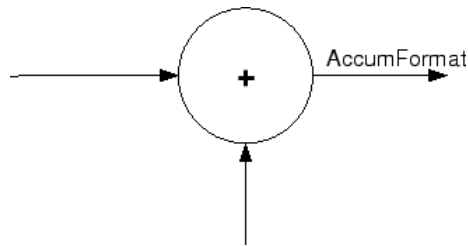
showing that the input data to the sum operations (the addends) retain their format word length and fraction length from previous operations. The addition process uses the existing input formats and then casts the output to the format defined by `AccumFormat`. Thus the output data has the word length and fraction length defined by `AccumWordLength` and `AccumFracLength`.

When `CastBeforeSum` is `true`, sum elements in filter signal flow diagrams look like this:
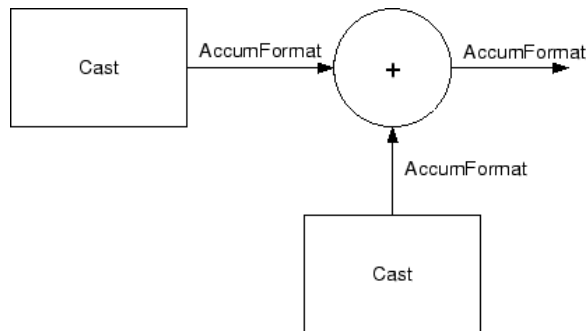
showing that the input data gets recast to the accumulator format word length and fraction length (AccumFormat) before the sum operation occurs. The data output by the addition operation has the word length and fraction length defined by `AccumWordLength` and `AccumFracLength`.

CoeffAutoScale

How the filter represents the filter coefficients depends on the property value of `CoeffAutoScale`. When you create a `dfilt` object, you use coefficients in double-precision format. Converting the `dfilt` object to fixed-point arithmetic forces the coefficients into a fixed-point representation. The representation the filter uses depends on whether the value of `CoeffAutoScale` is `true` or `false`.

- `CoeffAutoScale = true` means the filter chooses the fraction length to maintain the value of the coefficients as close to the double-precision values as possible. When you change the word length applied to the coefficients, the filter object changes the fraction length to try to accommodate the change. `true` is the default setting.

- `CoeffAutoScale = false` removes the automatic scaling of the fraction length for the coefficients and exposes the property that controls the coefficient fraction length so you can change it. For example, if the filter is a direct form FIR filter, setting `CoeffAutoScale = false` exposes the `NumFracLength` property that specifies the fraction length applied to numerator coefficients. If the filter is an IIR filter, setting `CoeffAutoScale = false` exposes both the `NumFracLength` and `DenFracLength` properties.

Here is an example of using `CoeffAutoScale` with a direct form filter.

```
hd2=dfilt.dffir([0.3 0.6 0.3])

hd2 =

        FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'double'
                Numerator: [0.3000 0.6000 0.3000]
        PersistentMemory: false
                  States: [2x1 double]
```

```
hd2.arithmetic='fixed'

hd2 =

          FilterStructure: 'Direct-Form FIR'
                Arithmetic: 'fixed'
                 Numerator: [0.3000 0.6000 0.3000]
          PersistentMemory: false
                    States: [1x1 embedded.fi]

           CoeffWordLength: 16
            CoeffAutoScale: true
                    Signed: true

           InputWordLength: 16
           InputFracLength: 15

          OutputWordLength: 16
                OutputMode: 'AvoidOverflow'

               ProductMode: 'FullPrecision'

           AccumWordLength: 40
             CastBeforeSum: true

                 RoundMode: 'convergent'
              OverflowMode: 'wrap'
```

To this point, the filter coefficients retain the original values from when you
created the filter as shown in the Numerator property. Now change the
CoeffAutoScale property value from true to false.

```
hd2.coeffautoScale=false

hd2 =

          FilterStructure: 'Direct-Form FIR'
                Arithmetic: 'fixed'
                 Numerator: [0.3000 0.6000 0.3000]
          PersistentMemory: false
```

```
              States: [1x1 embedded.fi]

    CoeffWordLength: 16
     CoeffAutoScale: false
       NumFracLength: 15
              Signed: true

    InputWordLength: 16
    InputFracLength: 15

   OutputWordLength: 16
         OutputMode: 'AvoidOverflow'

        ProductMode: 'FullPrecision'

    AccumWordLength: 40
       CastBeforeSum: true

          RoundMode: 'convergent'
       OverflowMode: 'wrap'
```

With the `NumFracLength` property now available, change the word length to 5 bits.

Notice the coefficient values. Setting `CoeffAutoScale` to `false` removes the automatic fraction length adjustment and the filter coefficients cannot be represented by the current format of [5 15] — a word length of 5 bits, fraction length of 15 bits.

```
hd2.coeffwordlength=5

hd2 =

        FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [4.5776e-004 4.5776e-004 4.5776e-004]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

        CoeffWordLength: 5
```

```
                    CoeffAutoScale: false
                      NumFracLength: 15
                             Signed: true

                  InputWordLength: 16
                  InputFracLength: 15

                 OutputWordLength: 16
                       OutputMode: 'AvoidOverflow'

                       ProductMode: 'FullPrecision'

                   AccumWordLength: 40
                     CastBeforeSum: true

                         RoundMode: 'convergent'
                      OverflowMode: 'wrap'
```

Restoring `CoeffAutoScale` to `true` goes some way to fixing the coefficient values. Automatically scaling the coefficient fraction length results in setting the fraction length to 4 bits. You can check this with `get(hd2)` as shown below.

```
hd2.coeffautoScale=true

hd2 =

             FilterStructure: 'Direct-Form FIR'
                  Arithmetic: 'fixed'
                   Numerator: [0.3125 0.6250 0.3125]
             PersistentMemory: false
                       States: [1x1 embedded.fi]

              CoeffWordLength: 5
              CoeffAutoScale: true
                       Signed: true

              InputWordLength: 16
              InputFracLength: 15
```

```
        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

             ProductMode: 'FullPrecision'

          AccumWordLength: 40
            CastBeforeSum: true

                RoundMode: 'convergent'
             OverflowMode: 'wrap'

get(hd2)
        PersistentMemory: false
FilterStructure: 'Direct-Form FIR'
                   States: [1x1 embedded.fi]
                Numerator: [0.3125 0.6250 0.3125]
               Arithmetic: 'fixed'
           CoeffWordLength: 5
            CoeffAutoScale: 1
                   Signed: 1
                RoundMode: 'convergent'
             OverflowMode: 'wrap'
          InputWordLength: 16
          InputFracLength: 15
         OutputWordLength: 16
               OutputMode: 'AvoidOverflow'
              ProductMode: 'FullPrecision'
            NumFracLength: 4
         OutputFracLength: 12
        ProductWordLength: 21
        ProductFracLength: 19
          AccumWordLength: 40
          AccumFracLength: 19
            CastBeforeSum: 1
```

Clearly five bits is not enough to represent the coefficients accurately.

CoeffFracLength

Fixed-point scalar filters that you create using `dfilt.scalar` use this property to define the fraction length applied to the scalar filter coefficients. Like the coefficient-fraction-length-related properties for the FIR, lattice, and IIR filters, `CoeffFracLength` is not displayed for scalar filters until you set `CoeffAutoScale` to `false`. Once you change the automatic scaling you can set the fraction length for the coefficients to any value you require.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 14 bits, with the `CoeffWordlength` of 16 bits.

CoeffWordLength

One primary consideration in developing filters for hardware is the length of a data word. `CoeffWordLength` defines the word length for these data storage and arithmetic locations:

- Numerator and denominator filter coefficients
- Tap sum in `dfilt.dfsymfir` and `dfilt.dfasymfir` filter objects
- Section input, multiplicand, and state values in direct-form SOS filter objects such as `dfilt.df1t` and `dfilt.df2`
- Scale values in second-order filters
- Lattice and ladder coefficients in lattice filter objects, such as `dfilt.latticearma` and `dfilt.latticemamax`
- Gain in `dfilt.scalar`

Setting this property value controls the word length for the data listed. In most cases, the data words in this list have separate fraction length properties to define the associated fraction lengths.

Any positive, integer word length works here, limited by the machine you use to develop your filter and the hardware you use to deploy your filter.

DenAccumFracLength

Filter structures `df1`, `df1t`, `df2`, and `df2t` that use `fixed` arithmetic have this property that defines the fraction length applied to denominator coefficients in the accumulator. In combination with `AccumWordLength`, the properties fully specify how the accumulator outputs data stored there.

As with all fraction length properties, `DenAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. To be able to change the property value for this property, you set `FilterInternals` to `SpecifyPrecision`.

DenFracLength

Property `DenFracLength` contains the value that specifies the fraction length for the denominator coefficients for your filter. `DenFracLength` specifies the fraction length used to interpret the data stored in `C`. Used in combination with `CoeffWordLength`, these two properties define the interpretation of the coefficients stored in the vector that contains the denominator coefficients.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `CoeffWordLength` of 16 bits.

Denominator

The denominator coefficients for your IIR filter, taken from the prototype you start with, are stored in this property. Generally this is a 1-by-N array of data in double format, where N is the length of the filter.

All IIR filter objects include `Denominator`, except the lattice-based filters which store their coefficients in the `Lattice` property, and second-order section filters, such as `dfilt.df1tsos`, which use the `SosMatrix` property to hold the coefficients for the sections.

DenProdFracLength

A property of all of the direct form IIR `dfilt` objects, except the ones that implement second-order sections, `DenProdFracLength` specifies the fraction

length applied to data output from product operations that the filter performs on denominator coefficients.

Looking at the signal flow diagram for the `dfilt.df1t` filter, for example, you see that denominators and numerators are handled separately. When you set `ProductMode` to `SpecifyPrecision`, you can change the `DenProdFracLength` setting manually. Otherwise, for multiplication operations that use the denominator coefficients, the filter sets the fraction length as defined by the `ProductMode` setting.

DenStateFracLength

When you look at the flow diagram for the `dfilt.df1sos` filter object, the states associated with denominator coefficient operations take the fraction length from this property. In combination with the `DenStateWordLength` property, these properties fully specify how the filter interprets the states.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `DenStateWordLength` of 16 bits.

DenStateWordLength

When you look at the flow diagram for the `dfilt.df1sos` filter object, the states associated with the denominator coefficient operations take the data format from this property and the `DenStateFracLength` property. In combination, these properties fully specify how the filter interprets the state it uses.

By default, the value is 16 bits, with the `DenStateFracLength` of 15 bits.

FilterInternals

Similar to the FilterInternals pane in FDATool, this property controls whether the filter sets the output word and fraction lengths automatically, and the accumulator word and fraction lengths automatically as well, to maintain the best precision results during filtering. The default value, `FullPrecision`, sets automatic word and fraction length determination by the filter. Setting `FilterInternals` to `SpecifyPrecision` exposes the output

and accumulator related properties so you can set your own word and fraction lengths for them. Note that

FilterStructure

Every `dfilt` object has a `FilterStructure` property. This is a read-only property containing a string that declares the structure of the filter object you created.

When you construct filter objects, the `FilterStructure` property value is returned containing one of the strings shown in the following table. Property `FilterStructure` indicates the filter architecture and comes from the constructor you use to create the filter.

After you create a filter object, you cannot change the `FilterStructure` property value. To make filters that use different structures, you construct new filters using the appropriate methods, or use `convert` to switch to a new structure.

Default value. Since this depends on the constructor you use and the constructor includes the filter structure definition, there is no default value. When you try to create a filter without specifying a structure, MATLAB returns an error.

Filter Constructor Name	FilterStructure Property String and Filter Type
`'dfilt.df1'`	Direct form I
`'dfilt.df1sos'`	Direct form I filter implemented using second-order sections
`'dfilt.df1t'`	Direct form I transposed
`'dfilt.df2'`	Direct form II
`'dfilt.df2sos'`	Direct form II filter implemented using second order sections
`'dfilt.df2t'`	Direct form II transposed
`'dfilt.dfasymfir'`	Antisymmetric finite impulse response (FIR). Even and odd forms.

Filter Constructor Name	FilterStructure Property String and Filter Type
`'dfilt.dffir'`	Direct form FIR
`'dfilt.dffirt'`	Direct form FIR transposed
`'dfilt.latticeallpass'`	Lattice allpass
`'dfilt.latticear'`	Lattice autoregressive (AR)
`'dfilt.latticemamin'`	Lattice moving average (MA) minimum phase
`'dfilt.latticemamax'`	Lattice moving average (MA) maximum phase
`'dfilt.latticearma'`	Lattice ARMA
`'dfilt.dfsymfir'`	Symmetric FIR. Even and odd forms
`'dfilt.scalar'`	Scalar

Filter Structures with Quantizations Shown in Place. To help you understand how and where the quantizations occur in filter structures in this toolbox, the figure below shows the structure for a Direct Form II filter, including the quantizations (fixed-point formats) that compose part of the fixed-point filter. You see that one or more quantization processes, specified by the *format label, accompany each filter element, such as a delay, product, or summation element. The input to or output from each element reflects the result of applying the associated quantization as defined by the word length and fraction length format. Wherever a particular filter element appears in a filter structure, recall the quantization process that accompanies the element as it appears in this figure. Each filter reference page, such as the dfilt.df2 reference page, includes the signal flow diagram showing the formatting elements that define the quantizations that occur throughout the filter flow.

For example, a product quantization, either numerator or denominator, follows every product (gain) element and a sum quantization, also either numerator or denominator, follows each sum element. The figure shows the Arithmetic property value set to fixed.

df2 IIR Filter Structure Including the Formatting Objects, with Arithmetic Property Value fixed
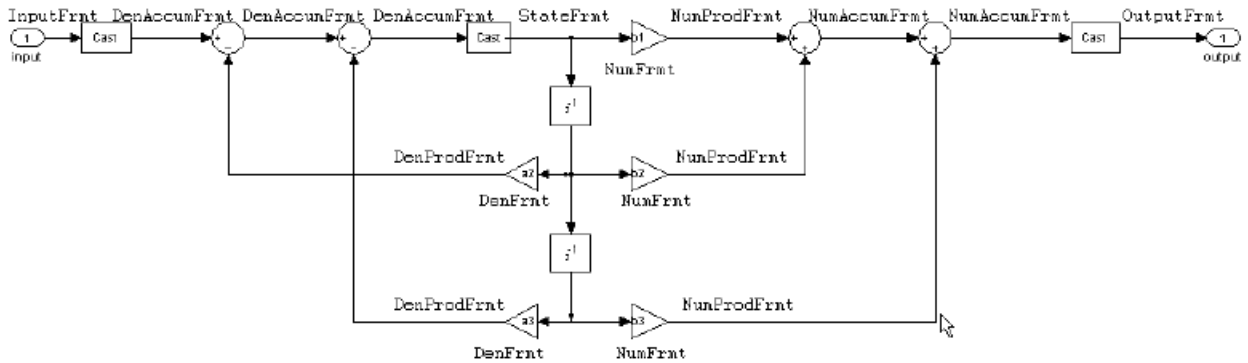
When your `df2` filter uses the `Arithmetic` property set to `fixed`, the filter structure contains the formatting features shown in the diagram. The formats included in the structure are fixed-point objects that include properties to set various word and fraction length formats. For example, the `NumFormat` or `DenFormat` in the fixed-point arithmetic filter set the properties for quantizing numerator or denominator coefficients according to word and fraction length settings.

When the leading denominator coefficient a(1) in your filter is not 1, choose it to be a power of two so that a shift replaces the multiply that would otherwise be used.

Fixed-Point Arithmetic Filter Structures. You choose among several filter structures when you create fixed-point filters. You can also specify filters with single or multiple cascaded sections of the same type. Because quantization is a nonlinear process, different fixed-point filter structures produce different results.

To specify the filter structure, you select the appropriate `dfilt.`*`structure`* method to construct your filter. Refer to the function reference information for `dfilt` and `set` for details on setting property values for quantized filters.

The figures in the following subsections of this section serve as aids to help you determine how to enter your filter coefficients for each filter structure. Each subsection contains an example for constructing a filter of the given structure.

Scale factors for the input and output for the filters do not appear in the block diagrams. The default filter structures do not include, nor assume, the scale factors. For filter scaling information, refer to `scale` in the Help system.

About the Filter Structure Diagrams. In the diagrams that accompany the following filter structure descriptions, you see the active operators that define the filter, such as sums and gains, and the formatting features that control the processing in the filter. Notice also that the coefficients are labeled in the figure. This tells you the order in which the filter processes the coefficients.

While the meaning of the block elements is straightforward, the labels for the formats that form part of the filter are less clear. Each figure includes text in the form *labelFormat* that represents the existence of a formatting feature at that point in the structure. The *Format* stands for formatting object and the *label* specifies the data that the formatting object affects.

For example, in the `dfilt.df2` filter shown above, the entries `InputFormat` and `OutputFormat` are the formats applied, that is the word length and fraction length, to the filter input and output data. For example, filter properties like `OutputWordLength` and `InputWordLength` specify values that control filter operations at the input and output points in the structure and are represented by the formatting objects `InputFormat` and `OutputFormat` shown in the filter structure diagrams.

Direct Form I Filter Structure. The following figure depicts the *direct form I* filter structure that directly realizes a transfer function with a second-order numerator and denominator. The numerator coefficients are numbered $b(i)$, i =1, 2, 3; the denominator coefficients are numbered $a(i)$, $i = 1$, 2, 3; and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the figure, the `Arithmetic` property is set to `fixed`.

Example — Specifying a Direct Form I Filter. You can specify a second-order direct form I structure for a quantized filter hq with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1(b,a);
```

To create the fixed-point filter, set the Arithmetic property to fixed as shown here.

```
set(hq,'arithmetic','fixed');
```

Direct Form I Filter Structure With Second-Order Sections. The following figure depicts a *direct form I* filter structure that directly realizes a transfer function with a second-order numerator and denominator and second-order sections. The numerator coefficients are numbered $b(i)$, i =1, 2, 3; the denominator coefficients are numbered $a(i)$, i = 1, 2, 3; and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the figure, the Arithmetic property is set to fixed to place the filter in fixed-point mode.

8-47

Example — Specifying a Direct Form I Filter with Second-Order Sections. You can specify an eighth-order direct form I structure for a quantized filter hq with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1sos(b,a);
```

To create the fixed-point filter, set the Arithmetic property to fixed, as shown here.

```
set(hq,'arithmetic','fixed');
```

Direct Form I Transposed Filter Structure. The next signal flow diagram depicts a *direct form I transposed* filter structure that directly realizes a transfer function with a second-order numerator and denominator. The numerator coefficients are $b(i)$, $i = 1, 2, 3$; the denominator coefficients are $a(i)$, $i = 1, 2, 3$; and the states (used for initial and final state values in filtering) are labeled $z(i)$. With the Arithmetic property value set to fixed, the figure shows the filter with the properties indicated.
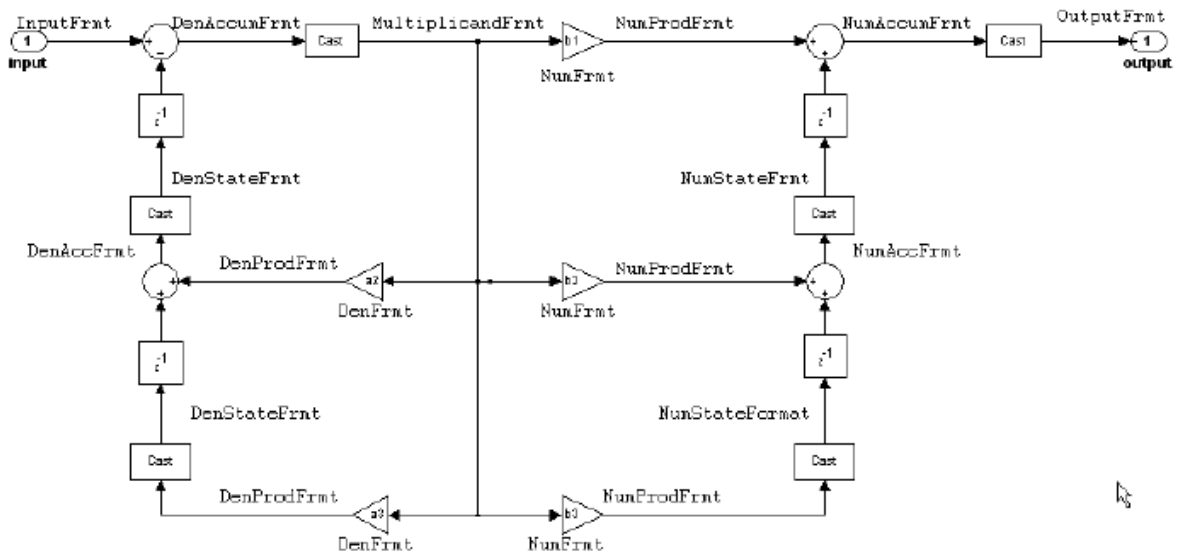
Example — Specifying a Direct Form I Transposed Filter. You can specify a second-order direct form I transposed filter structure for a quantized filter `hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df1t(b,a);
set(hq,'arithmetic','fixed');
```

Direct Form II Filter Structure. The following graphic depicts a *direct form II* filter structure that directly realizes a transfer function with a second-order numerator and denominator. In the figure, the `Arithmetic` property value is `fixed`. Numerator coefficients are named $b(i)$; denominator coefficients are named $a(i)$, $i = 1, 2, 3$; and the states (used for initial and final state values in filtering) are named $z(i)$.

Use the method `dfilt.df2` to construct a quantized filter whose
`FilterStructure` property is `Direct-Form II`.

Example — Specifying a Direct Form II Filter. You can specify a
second-order direct form II filter structure for a quantized filter `hq` with the
following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df2(b,a);
hq.arithmetic = 'fixed'
```

To convert your initial double-precision filter `hq` to a quantized or fixed-point
filter, set the `Arithmetic` property to `fixed`, as shown.

Direct Form II Filter Structure With Second-Order Sections

The following figure depicts *direct form II* filter structure using second-order
sections that directly realizes a transfer function with a second-order
numerator and denominator sections. In the figure, the `Arithmetic` property
value is `fixed`. Numerator coefficients are labeled $b(i)$; denominator
coefficients are labeled $a(i)$, $i = 1, 2, 3$; and the states (used for initial and final
state values in filtering) are labeled $z(i)$.

Use the method `dfilt.df2sos` to construct a quantized filter whose `FilterStructure` property is `Direct-Form II`.

Example — Specifying a Direct Form II Filter with Second-Order Sections. You can specify a tenth-order direct form II filter structure that uses second-order sections for a quantized filter `hq` with the following code.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hq = dfilt.df2sos(b,a);
hq.arithmetic = 'fixed'
```

To convert your prototype double-precision filter `hq` to a fixed-point filter, set the `Arithmetic` property to `fixed`, as shown.

Direct Form II Transposed Filter Structure. The following figure depicts the *direct form II transposed* filter structure that directly realizes transfer functions with a second-order numerator and denominator. The numerator coefficients are labeled $b(i)$, the denominator coefficients are labeled $a(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are labeled $z(i)$. In the first figure, the `Arithmetic` property value is `fixed`.

Use the constructor `dfilt.df2t` to specify the value of the `FilterStructure` property for a filter with this structure that you can convert to fixed-point filtering.

Example — Specifying a Direct Form II Transposed Filter. Specifying or constructing a second-order direct form II transposed filter for a fixed-point filter `hq` starts with the following code to define the coefficients and construct the filter.

```
b = [0.3 0.6 0.3];
a = [1 0 0.2];
hd = dfilt.df2t(b,a);
```

Now create the fixed-point filtering version of the filter from `hd`, which is floating point.

```
hq = set(hd,'arithmetic','fixed');
```

Direct Form Antisymmetric FIR Filter Structure (Any Order). The following figure depicts a *direct form antisymmetric FIR* filter structure that directly realizes a second-order antisymmetric FIR filter. The filter coefficients are labeled *b(i)*, and the initial and final state values in filtering are labeled *z*(*i*). This structure reflects the Arithmetic property set to fixed.
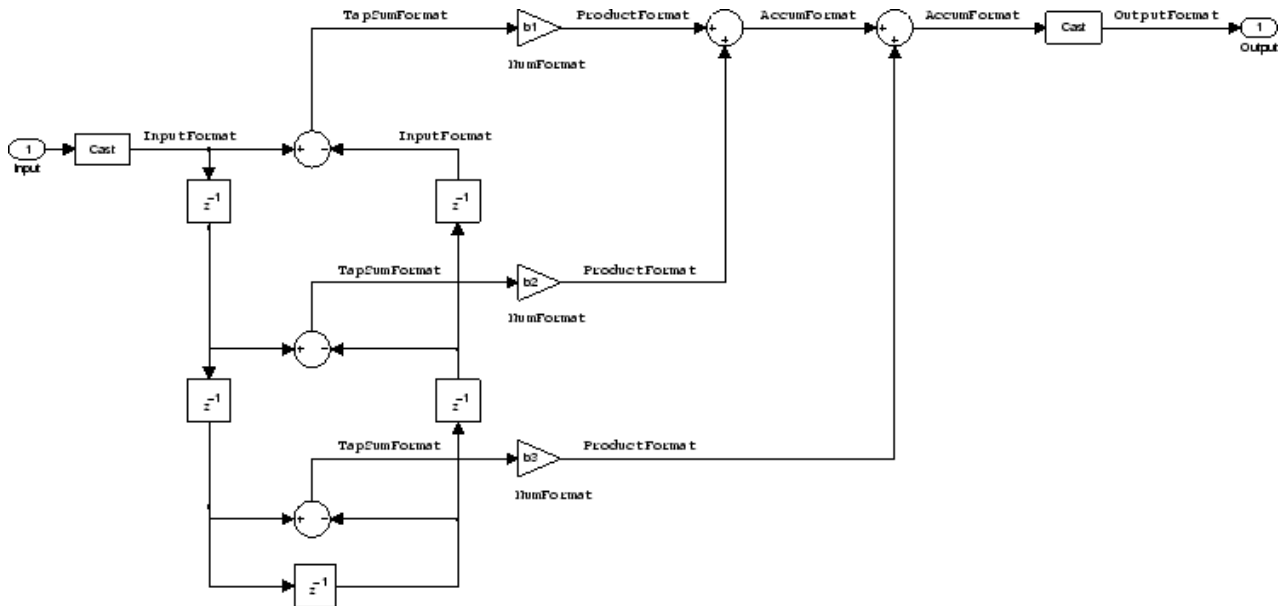
Use the method dfilt.dfasymfir to construct the filter, and then set the Arithmetic property to fixed to convert to a fixed-point filter with this structure.

Example — Specifying an Odd-Order Direct Form Antisymmetric FIR Filter. Specify a fifth-order direct form antisymmetric FIR filter structure for a fixed-point filter hq with the following code.

```
b = [-0.008 0.06 -0.44 0.44 -0.06 0.008];
hq = dfilt.dfasymfir(b);
set(hq,'arithmetic','fixed')

hq
```

```
hq =

        FilterStructure: 'Direct-Form Antisymmetric FIR'
             Arithmetic: 'fixed'
              Numerator: [-0.0080 0.0600 -0.4400 0.4400 -0.0600 0.0080]
       PersistentMemory: false
                 States: [1x1 fi object]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

              TapSumMode: 'KeepMSB'
        TapSumWordLength: 17

             ProductMode: 'FullPrecision'

         AccumWordLength: 40

           CastBeforeSum: true
               RoundMode: 'convergent'
            OverflowMode: 'wrap'

        InheritSettings: false
```

Example — Specifying an Even-Order Direct Form Antisymmetric FIR Filter. You can specify a fourth-order direct form antisymmetric FIR filter structure for a fixed-point filter hq with the following code.

```
b = [-0.01 0.1 0.0 -0.1 0.01];
hq = dfilt.dfasymfir(b);
hq.arithmetic='fixed'

hq =
```

```
   FilterStructure: 'Direct-Form Antisymmetric FIR'
        Arithmetic: 'fixed'
         Numerator: [-0.0100 0.1000 0 -0.1000 0.0100]
  PersistentMemory: false
            States: [1x1 fi object]

  CoeffWordLength: 16
   CoeffAutoScale: true
           Signed: true

  InputWordLength: 16
  InputFracLength: 15

 OutputWordLength: 16
       OutputMode: 'AvoidOverflow'

       TapSumMode: 'KeepMSB'
 TapSumWordLength: 17

      ProductMode: 'FullPrecision'

  AccumWordLength: 40

    CastBeforeSum: true
        RoundMode: 'convergent'
     OverflowMode: 'wrap'

  InheritSettings: false
```

Direct Form Finite Impulse Response (FIR) Filter Structure. In the next figure, you see the signal flow graph for a *direct form finite impulse response (FIR)* filter structure that directly realizes a second-order FIR filter. The filter coefficients are $b(i)$, $i = 1, 2, 3$, and the states (used for initial and final state values in filtering) are $z(i)$. To generate the figure, set the Arithmetic property to fixed after you create your prototype filter in double-precision arithmetic.
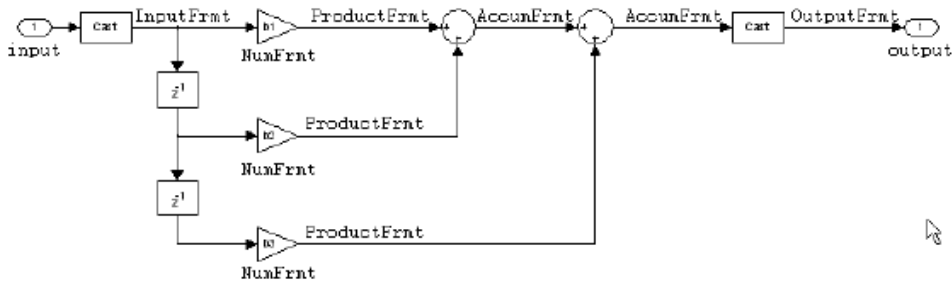
Use the `dfilt.dffir` method to generate a filter that uses this structure.

Example — Specifying a Direct Form FIR Filter. You can specify a second-order direct form FIR filter structure for a fixed-point filter `hq` with the following code.

```
b = [0.05 0.9 0.05];
hd = dfilt.dffir(b);
hq = set(hd,'arithmetic','fixed');
```

Direct Form FIR Transposed Filter Structure. This figure uses the filter coefficients labeled $b(i)$, i = 1, 2, 3, and states (used for initial and final state values in filtering) are labeled $z(i)$. These depict a *direct form finite impulse response (FIR) transposed* filter structure that directly realizes a second-order FIR filter.
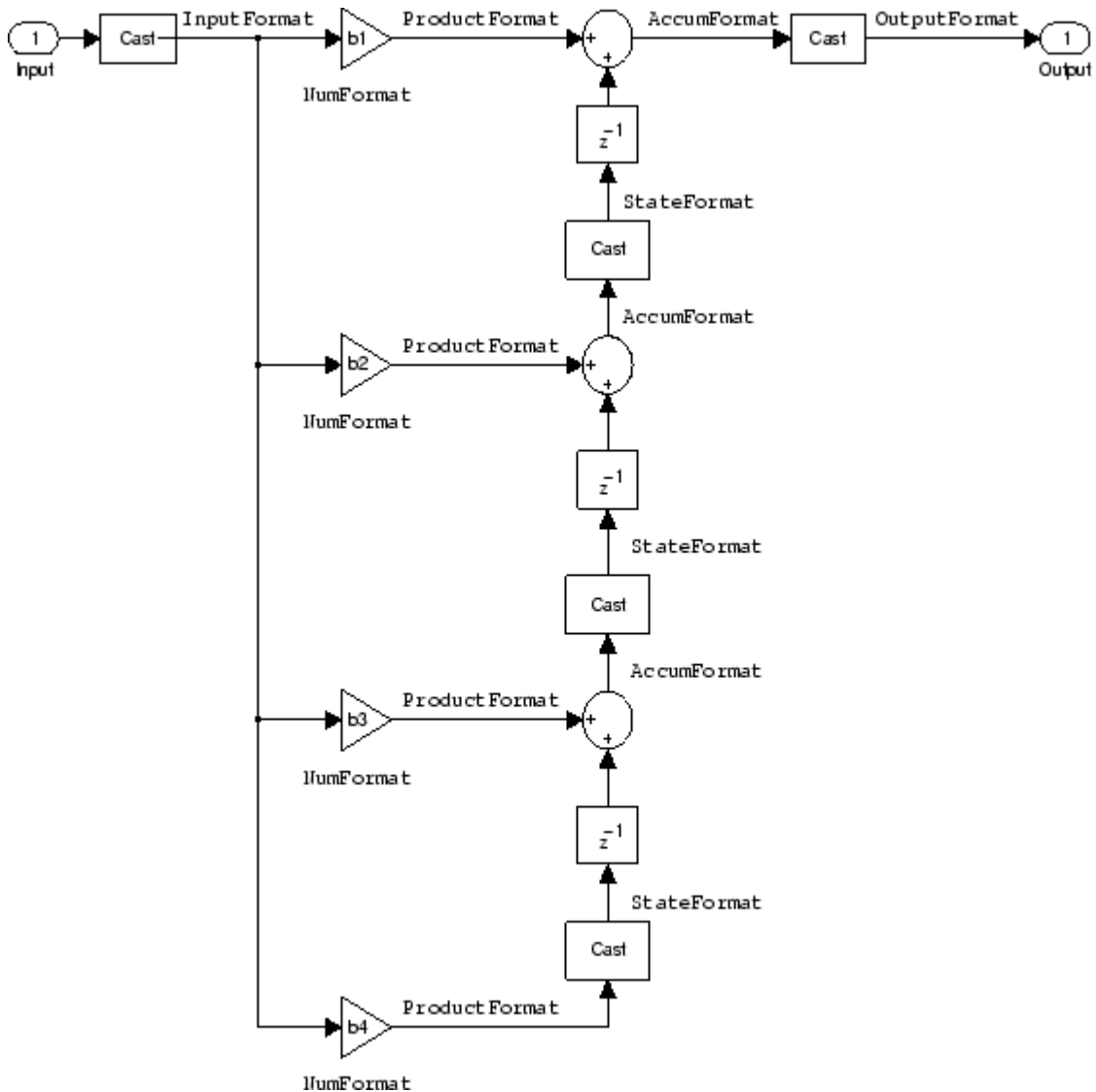
With the `Arithmetic` property set to `fixed`, your filter matches the figure. Using the method `dfilt.dffirt` returns a double-precision filter that you convert to a fixed-point filter.

Example — Specifying a Direct Form FIR Transposed Filter. You can specify a second-order direct form FIR transposed filter structure for a fixed-point filter hq with the following code.

```
b = [0.05 0.9 0.05];
hd=dfilt.dffirt(b);
hq = copy(hd);
hq.arithmetic = 'fixed';
```

Lattice Allpass Filter Structure. The following figure depicts the *lattice allpass* filter structure. The pictured structure directly realizes third-order lattice allpass filters using fixed-point arithmetic. The filter reflection coefficients are labeled $k1(i)$, $i = 1, 2, 3$. The states (used for initial and final state values in filtering) are labeled $z(i)$.
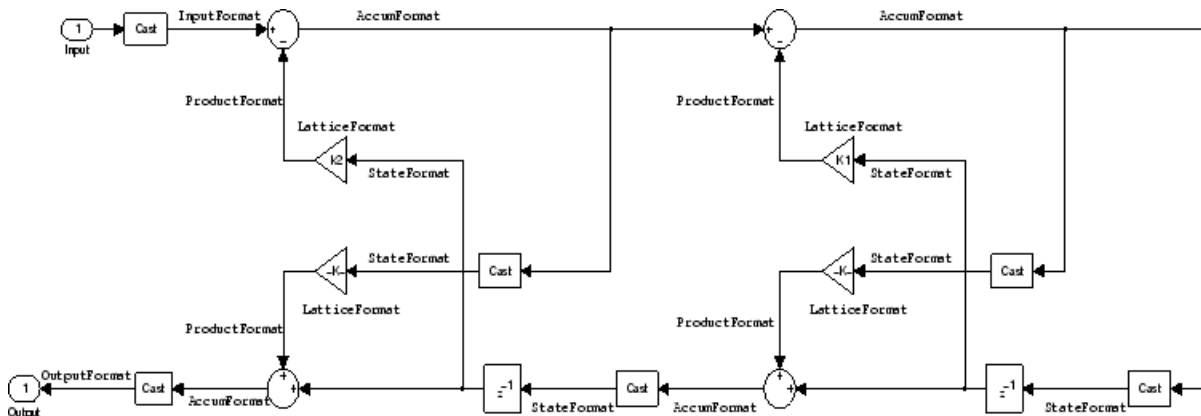
To create a quantized filter that uses the lattice allpass structure shown in the figure, use the dfilt.latticeallpass method and set the Arithmetic property to fixed.

Example — Specifying a Lattice Allpass Filter. You can create a third-order lattice allpass filter structure for a quantized filter hq with the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticeallpass(k);
set(hq,'arithmetic','fixed');
```

Lattice Moving Average Maximum Phase Filter Structure. In the next figure you see a *lattice moving average maximum phase* filter structure. This signal flow diagram directly realizes a third-order lattice moving average (MA) filter with the following phase form depending on the initial transfer function:

- When you start with a minimum phase transfer function, the upper branch of the resulting lattice structure returns a minimum phase filter. The lower branch returns a maximum phase filter.

- When your transfer function is neither minimum phase nor maximum phase, the lattice moving average maximum phase structure will not be maximum phase.

- When you start with a maximum phase filter, the resulting lattice filter is maximum phase also.

The filter reflection coefficients are labeled $k(i)$, $i = 1, 2, 3$. The states (used for initial and final state values in filtering) are labeled $z(i)$. In the figure, we set the Arithmetic property to fixed to reveal the fixed-point arithmetic format features that control such options as word length and fraction length.
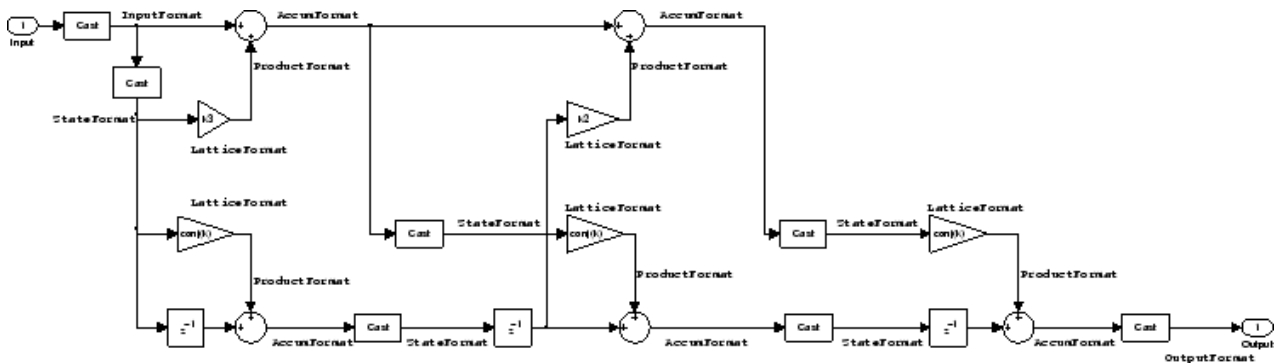
Example — Constructing a Lattice Moving Average Maximum Phase Filter. Constructing a fourth-order lattice MA maximum phase filter structure for a quantized filter hq begins with the following code.

```
k = [.66 .7 .44 .33];
hd=dfilt.latticemamax(k);
```

Lattice Autoregressive (AR) Filter Structure. The method
`dfilt.latticear` directly realizes lattice autoregressive filters in the toolbox.
The following figure depicts the third-order *lattice autoregressive (AR)*
filter structure — with the `Arithmetic` property equal to `fixed`. The filter
reflection coefficients are labeled $k(i)$, i = 1, 2, 3, and the states (used for
initial and final state values in filtering) are labeled $z(i)$.
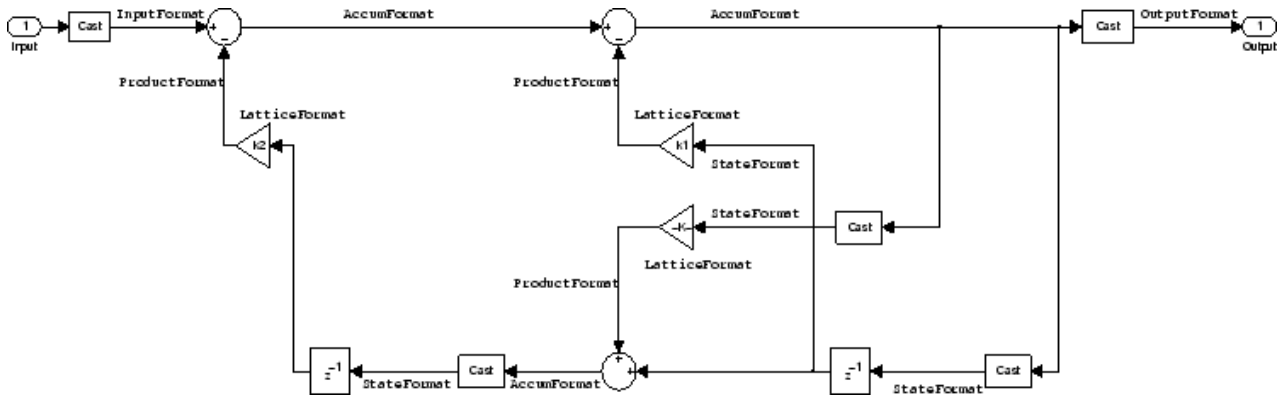
Example — Specifying a Lattice AR Filter. You can specify a third-order
lattice AR filter structure for a quantized filter `hq` with the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticear(k);
hq.arithmetic = 'custom';
```

Lattice Moving Average (MA) Filter Structure for Minimum Phase.
The following figures depict *lattice moving average (MA)* filter structures
that directly realize third-order lattice MA filters for minimum phase. The
filter reflection coefficients are labeled $k(i)$, (i). = 1, 2, 3, and the states (used
for initial and final state values in filtering) are labeled $z(i)$. Setting the
`Arithmetic` property of the filter to `fixed` results in a fixed-point filter that
matches the figure.

This signal flow diagram directly realizes a third-order lattice moving average
(MA) filter with the following phase form depending on the initial transfer
function:

- When you start with a minimum phase transfer function, the upper branch of the resulting lattice structure returns a minimum phase filter. The lower branch returns a minimum phase filter.

- When your transfer function is neither minimum phase nor maximum phase, the lattice moving average minimum phase structure will not be minimum phase.

- When you start with a minimum phase filter, the resulting lattice filter is minimum phase also.

The filter reflection coefficients are labeled $k((i).)$, $i = 1, 2, 3$. The states (used for initial and final state values in filtering) are labeled $z((i).)$. This figure shows the filter structure when the Arithmetic property is set to fixed to reveal the fixed-point arithmetic format features that control such options as word length and fraction length.
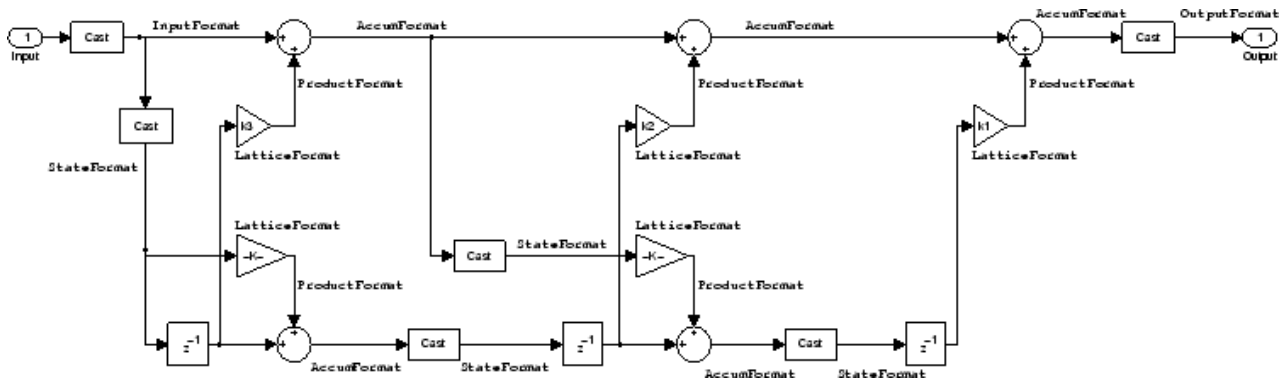
Example — Specifying a Minimum Phase Lattice MA Filter. You can specify a third-order lattice MA filter structure for minimum phase applications using variations of the following code.

```
k = [.66 .7 .44];
hd=dfilt.latticemamin(k);
set(hq,'arithmetic','fixed');
```

Lattice Autoregressive Moving Average (ARMA) Filter Structure. The figure below depicts a *lattice autoregressive moving average (ARMA)* filter structure that directly realizes a fourth-order lattice ARMA filter. The filter reflection coefficients are labeled $k(i)$, (i). = 1, ..., 4; the ladder coefficients are labeled $v(i)$, (i). = 1, 2, 3; and the states (used for initial and final state values in filtering) are labeled $z(i)$.
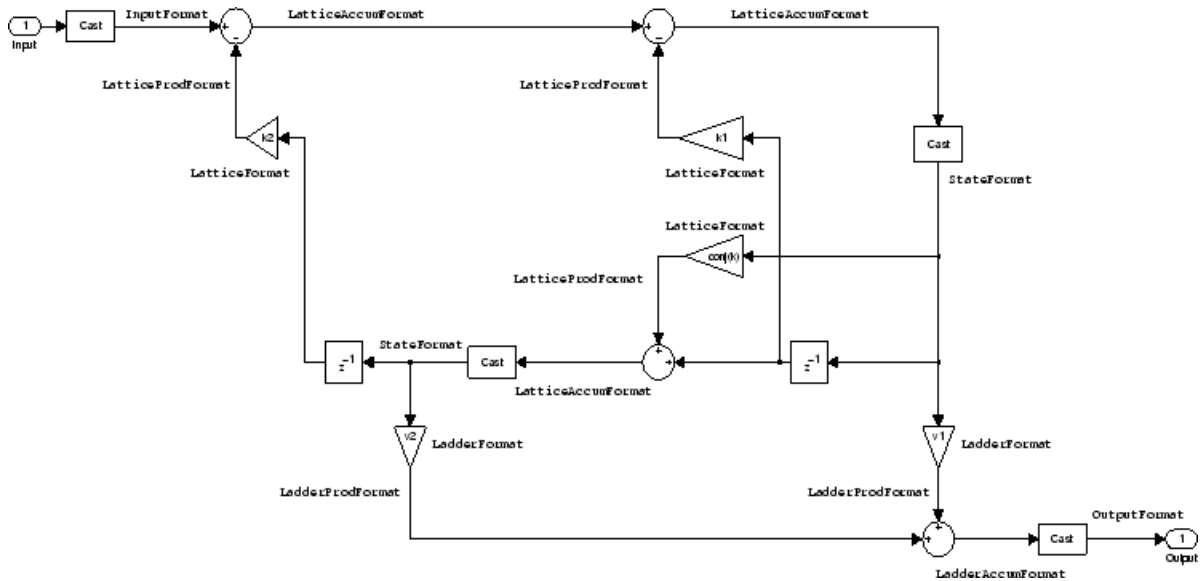
Example — Specifying an Lattice ARMA Filter. The following code specifies a fourth-order lattice ARMA filter structure for a quantized filter hq, starting from hd, a floating-point version of the filter.

```
k = [.66 .7 .44 .66];
v = [1 0 0];
hd=dfilt.latticearma(k,v);
hq.arithmetic = 'fixed';
```

Direct Form Symmetric FIR Filter Structure (Any Order). Shown in the next figure, you see signal flow that depicts a *direct form symmetric FIR* filter structure that directly realizes a fifth-order direct form symmetric FIR filter. Filter coefficients are labeled $b(i)$, $i = 1, ..., n$, and states (used for initial and final state values in filtering) are labeled $z(i)$. Showing the filter structure used when you select fixed for the Arithmetic property value, the first figure details the properties in the filter object.
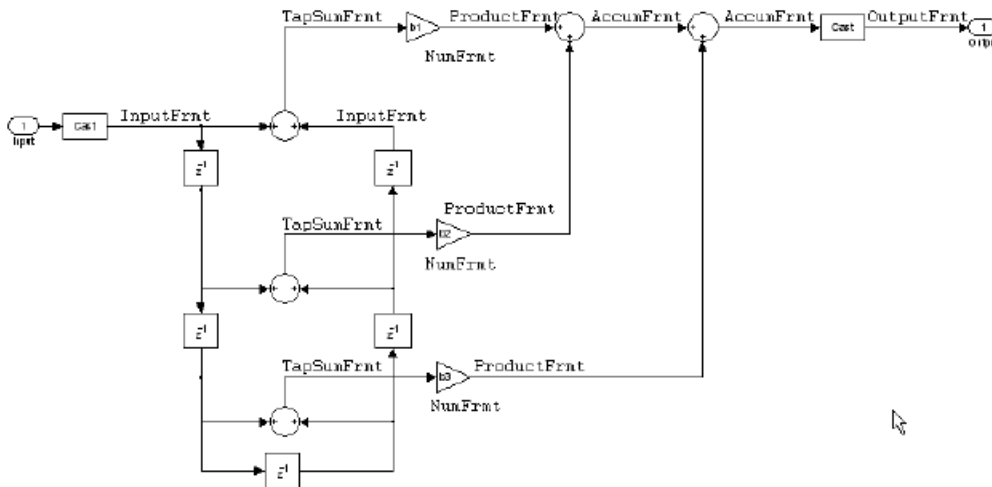
Example — Specifying an Odd-Order Direct Form Symmetric FIR Filter. By using the following code in MATLAB, you can specify a fifth-order direct form symmetric FIR filter for a fixed-point filter hq:

```
b = [-0.008 0.06 0.44 0.44 0.06 -0.008];
hd=dfilt.dfsymfir(b);
set(hq,'arithmetic','fixed');
```

Assigning Filter Coefficients. The syntax you use to assign filter coefficients for your floating-point or fixed-point filter depends on the structure you select for your filter.

Converting Filters Between Representations. Filter conversion functions in this toolbox and in Signal Processing Toolbox software let you convert filter transfer functions to other filter forms, and from other filter forms to transfer function form. Relevant conversion functions include the following functions.

Conversion Function	Description
ca2tf	Converts from a coupled allpass filter to a transfer function.
cl2tf	Converts from a lattice coupled allpass filter to a transfer function.
convert	Convert a discrete-time filter from one filter structure to another.
sos	Converts quantized filters to create second-order sections. We recommend this method for converting quantized filters to second-order sections.
tf2ca	Converts from a transfer function to a coupled allpass filter.
tf2cl	Converts from a transfer function to a lattice coupled allpass filter.
tf2latc	Converts from a transfer function to a lattice filter.
tf2sos	Converts from a transfer function to a second-order section form.
tf2ss	Converts from a transfer function to state-space form.
tf2zp	Converts from a rational transfer function to its factored (single section) form (zero-pole-gain form).
zp2sos	Converts a zero-pole-gain form to a second-order section form.
zp2ss	Conversion of zero-pole-gain form to a state-space form.
zp2tf	Conversion of zero-pole-gain form to transfer functions of multiple order sections.

Note that these conversion routines do not apply to `dfilt` objects.

The function `convert` is a special case — when you use convert to change
the filter structure of a fixed-point filter, you lose all of the filter states and
settings. Your new filter has default values for all properties, and it in not
fixed-point.

To demonstrate the changes that occur, convert a fixed-point direct form I
transposed filter to direct form II structure.

```
hd=dfilt.df1t

hd =

         FilterStructure: 'Direct-Form I Transposed'
              Arithmetic: 'double'
                Numerator: 1
              Denominator: 1
        PersistentMemory: false
                   States: Numerator:  [0x0 double]
                           Denominator:[0x0 double]


hd.arithmetic='fixed'
hd =

         FilterStructure: 'Direct-Form I Transposed'
              Arithmetic: 'fixed'
                Numerator: 1
              Denominator: 1
        PersistentMemory: false
                   States: Numerator:  [0x0 fi]
                           Denominator:[0x0 fi]

convert(hd,'df2')

Warning: Using reference filter for structure conversion.
Fixed-point attributes will not be converted.

ans =
```

```
     FilterStructure: 'Direct-Form II'
          Arithmetic: 'double'
           Numerator: 1
         Denominator: 1
    PersistentMemory: false
              States: [0x1 double]
```

You can specify a filter with *L* sections of arbitrary order by

1 Factoring your entire transfer function with `tf2zp`. This converts your transfer function to zero-pole-gain form.

2 Using `zp2tf` to compose the transfer function for each section from the selected first-order factors obtained in step 1.

Note You are not required to normalize the leading coefficients of each section's denominator polynomial when you specify second-order sections, though `tf2sos` does.

Gain
`dfilt.scalar` filters have a gain value stored in the `gain` property. By default the gain value is one — the filter acts as a wire.

InputFracLength
`InputFracLength` defines the fraction length assigned to the input data for your filter. Used in tandem with `InputWordLength`, the pair defines the data format for input data you provide for filtering.

As with all fraction length properties in `dfilt` objects, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, in this case `InputWordLength`, as well.

InputWordLength
Specifies the number of bits your filter uses to represent your input data. Your word length option is limited by the arithmetic you choose — up to 32 bits for

double, `float`, and `fixed`. Setting `Arithmetic` to `single` (single-precision floating-point) limits word length to 16 bits. The default value is 16 bits.

Ladder

Included as a property in `dfilt.latticearma` filter objects, `Ladder` contains the denominator coefficients that form an IIR lattice filter object. For instance, the following code creates a high pass filter object that uses the lattice ARMA structure.

```
[b,a]=cheby1(5,.5,.5,'high')

b =

    0.0282   -0.1409    0.2817   -0.2817    0.1409   -0.0282


a =

    1.0000    0.9437    1.4400    0.9629    0.5301    0.1620

hd=dfilt.latticearma(b,a)

hd =

        FilterStructure: [1x44 char]
              Arithmetic: 'double'
                  Lattice: [1x6 double]
                   Ladder: [1 0.9437 1.4400 0.9629 0.5301 0.1620]
        PersistentMemory: false
                   States: [6x1 double]

hd.arithmetic='fixed'

hd =

        FilterStructure: [1x44 char]
              Arithmetic: 'fixed'
                  Lattice: [1x6 double]
                   Ladder: [1 0.9437 1.4400 0.9629 0.5301 0.1620]
```

```
      PersistentMemory: false
                States: [1x1 embedded.fi]

      CoeffWordLength: 16
       CoeffAutoScale: true
               Signed: true

      InputWordLength: 16
      InputFracLength: 15

     OutputWordLength: 16
           OutputMode: 'AvoidOverflow'

      StateWordLength: 16
      StateFracLength: 15

          ProductMode: 'FullPrecision'

      AccumWordLength: 40
         CastBeforeSum: true

            RoundMode: 'convergent'
         OverflowMode: 'wrap'
```

LadderAccumFracLength

Autoregressive, moving average lattice filter objects (lattticearma) use ladder coefficients to define the filter. In combination with LadderFracLength and CoeffWordLength, these three properties specify or reflect how the accumulator outputs data stored there. As with all fraction length properties, LadderAccumFracLength can be any integer, including integers larger than AccumWordLength, and positive or negative integers. The default value is 29 bits.

LadderFracLength

To let you control the way your latticearma filter interprets the denominator coefficients, LadderFracLength sets the fraction length applied to the ladder coefficients for your filter. The default value is 14 bits.

As with all fraction length properties, `LadderFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers.

Lattice

When you create a lattice-based IIR filter, your numerator coefficients (from your IIR prototype filter or the default `dfilt` lattice filter function) get stored in the `Lattice` property of the `dfilt` object. The properties `CoeffWordLength` and `LatticeFracLength` define the data format the object uses to represent the lattice coefficients. By default, lattice coefficients are in double-precision format.

LatticeAccumFracLength

Lattice filter objects (`latticeallpass`, `latticearma`, `latticemamax`, and `latticemamin`) use lattice coefficients to define the filter. In combination with `LatticeFracLength` and `CoeffWordLength`, these three properties specify how the accumulator outputs lattice coefficient-related data stored there. As with all fraction length properties, `LatticeAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. By default, the property is set to 31 bits.

LatticeFracLength

To let you control the way your filter interprets the denominator coefficients, `LatticeFracLength` sets the fraction length applied to the lattice coefficients for your lattice filter. When you create the default lattice filter, `LatticeFracLength` is 16 bits.

As with all fraction length properties, `LatticeFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers.

MultiplicandFracLength

Each input data element for a multiply operation has both word length and fraction length to define its representation. `MultiplicandFracLength` sets the fraction length to use when the filter object performs any multiply operation during filtering. For default filters, this is set to 15 bits.

As with all word and fraction length properties, `MultiplicandFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers.

MultiplicandWordLength

Each input data element for a multiply operation has both word length and fraction length to define its representation. `MultiplicandWordLength` sets the word length to use when the filter performs any multiply operation during filtering. For default filters, this is set to 16 bits. Only the `df1t` and `df1tsos` filter objects include the `MultiplicandFracLength` property.

Only the `df1t` and `df1tsos` filter objects include the `MultiplicandWordLength` property.

NumAccumFracLength

Filter structures `df1`, `df1t`, `df2`, and `df2t` that use `fixed` arithmetic have this property that defines the fraction length applied to numerator coefficients in output from the accumulator. In combination with `AccumWordLength`, the `NumAccumFracLength` property fully specifies how the accumulator outputs numerator-related data stored there.

As with all fraction length properties, `NumAccumFracLength` can be any integer, including integers larger than `AccumWordLength`, and positive or negative integers. 30 bits is the default value when you create the filter object. To be able to change the value for this property, set `FilterInternals` for the filter to `SpecifyPrecision`.

Numerator

The numerator coefficients for your filter, taken from the prototype you start with or from the default filter, are stored in this property. Generally this is a 1-by-N array of data in double format, where N is the length of the filter.

All of the filter objects include `Numerator`, except the lattice-based and second-order section filters, such as `dfilt.latticema` and `dfilt.df1tsos`.

NumFracLength

Property `NumFracLength` contains the value that specifies the fraction length for the numerator coefficients for your filter. `NumFracLength` specifies the fraction length used to interpret the numerator coefficients. Used in combination with `CoeffWordLength`, these two properties define the interpretation of the coefficients stored in the vector that contains the numerator coefficients.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 15 bits, with the `CoeffWordLength` of 16 bits.

NumProdFracLength

A property of all of the direct form IIR `dfilt` objects, except the ones that implement second-order sections, `NumProdFracLength` specifies the fraction length applied to data output from product operations the filter performs on numerator coefficients.

Looking at the signal flow diagram for the `dfilt.df1t` filter, for example, you see that denominators and numerators are handled separately. When you set `ProductMode` to `SpecifyPrecision`, you can change the `NumProdFracLength` setting manually. Otherwise, for multiplication operations that use the numerator coefficients, the filter sets the word length as defined by the `ProductMode` setting.

NumStateFracLength

All the variants of the direct form I structure include the property `NumStateFracLength` to store the fraction length applied to the numerator states for your filter object. By default, this property has the value 15 bits, with the `CoeffWordLength` of 16 bits, which you can change after you create the filter object.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well.

NumStateWordLength

When you look at the flow diagram for the `df1sos` filter object, the states associated with the numerator coefficient operations take the data format from this property and the `NumStateFracLength` property. In combination, these properties fully specify how the filter interprets the state it uses.

As with all fraction length properties, the value you enter here can be any negative or positive integer, or zero. Fraction length can be larger than the associated word length, as well. By default, the value is 16 bits, with the `NumStateFracLength` of 11 bits.

OutputFracLength

To define the output from your filter object, you need both the word and fraction lengths. `OutputFracLength` determines the fraction length applied to interpret the output data. Combining this with `OutputWordLength` fully specifies the format of the output.

Your fraction length can be any negative or positive integer, or zero. In addition, the fraction length you specify can be larger than the associated word length. Generally, the default value is 11 bits.

OutputMode

Sets the mode the filter uses to scale the filtered (output) data. You have the following choices:

- `AvoidOverflow` — directs the filter to set the property that controls the output data fraction length to avoid causing the data to overflow. In a `df2` filter, this would be the `OutputFracLength` property.

- `BestPrecision` — directs the filter to set the property that controls the output data fraction length to maximize the precision in the output data. For `df1t` filters, this is the `OutputFracLength` property. When you change the word length (`OutputWordLength`), the filter adjusts the fraction length to maintain the best precision for the new word size.

- `SpecifyPrecision` — lets you set the fraction length used by the filtered data. When you select this choice, you can set the output fraction length using the `OutputFracLength` property to define the output precision.

All filters include this property except the direct form I filter which takes the output format from the filter states.

Here is an example that changes the mode setting to bestprecision, and then adjusts the word length for the output.

```
hd=dfilt.df2

hd =

         FilterStructure: 'Direct-Form II'
               Arithmetic: 'double'
                Numerator: 1
              Denominator: 1
         PersistentMemory: false
                   States: [0x1 double]

hd.arithmetic='fixed'

hd =

         FilterStructure: 'Direct-Form II'
               Arithmetic: 'fixed'
                Numerator: 1
              Denominator: 1
         PersistentMemory: false
                   States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

         StateWordLength: 16
         StateFracLength: 15
```

```
                        ProductMode: 'FullPrecision'

                    AccumWordLength: 40
                      CastBeforeSum: true

                          RoundMode: 'convergent'
                       OverflowMode: 'wrap'

get(hd)
                   PersistentMemory: false
                   FilterStructure: 'Direct-Form II'
                             States: [1x1 embedded.fi]
                          Numerator: 1
                        Denominator: 1
                         Arithmetic: 'fixed'
                     CoeffWordLength: 16
                      CoeffAutoScale: 1
                             Signed: 1
                          RoundMode: 'convergent'
                       OverflowMode: 'wrap'
                    InputWordLength: 16
                    InputFracLength: 15
                   OutputWordLength: 16
                         OutputMode: 'AvoidOverflow'
                        ProductMode: 'FullPrecision'
                    StateWordLength: 16
                    StateFracLength: 15
                        NumFracLength: 14
                        DenFracLength: 14
                    OutputFracLength: 13
                  ProductWordLength: 32
                  NumProdFracLength: 29
                  DenProdFracLength: 29
                    AccumWordLength: 40
                 NumAccumFracLength: 29
                 DenAccumFracLength: 29
                      CastBeforeSum: 1

hd.outputMode='bestprecision'
```

```
hd =

        FilterStructure: 'Direct-Form II'
             Arithmetic: 'fixed'
              Numerator: 1
            Denominator: 1
       PersistentMemory: false
                 States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: true
                  Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'BestPrecision'

         StateWordLength: 16
         StateFracLength: 15

             ProductMode: 'FullPrecision'

         AccumWordLength: 40
           CastBeforeSum: true

               RoundMode: 'convergent'
            OverflowMode: 'wrap'

hd.outputWordLength=8;

get(hd)
       PersistentMemory: false
        FilterStructure: 'Direct-Form II'
                 States: [1x1 embedded.fi]
              Numerator: 1
            Denominator: 1
             Arithmetic: 'fixed'
```

```
         CoeffWordLength: 16
          CoeffAutoScale: 1
                  Signed: 1
               RoundMode: 'convergent'
            OverflowMode: 'wrap'
         InputWordLength: 16
         InputFracLength: 15
        OutputWordLength: 8
              OutputMode: 'BestPrecision'
             ProductMode: 'FullPrecision'
         StateWordLength: 16
         StateFracLength: 15
           NumFracLength: 14
           DenFracLength: 14
        OutputFracLength: 5
       ProductWordLength: 32
       NumProdFracLength: 29
       DenProdFracLength: 29
         AccumWordLength: 40
      NumAccumFracLength: 29
      DenAccumFracLength: 29
           CastBeforeSum: 1
```

Changing the `OutputWordLength` to 8 bits caused the filter to change the `OutputFracLength` to 5 bits to keep the best precision for the output data.

OutputWordLength

Use the property `OutputWordLength` to set the word length used by the output from your filter. Set this property to a value that matches your intended hardware. For example, some digital signal processors use 32-bit output so you would set `OutputWordLength` to 32.

```
[b,a] = butter(6,.5);
hd=dfilt.df1t(b,a);

set(hd,'arithmetic','fixed')

hd

hd =
```

```
          FilterStructure: 'Direct-Form I Transposed'
               Arithmetic: 'fixed'
                Numerator: [1x7 double]
              Denominator: [1 0 0.7777 0 0.1142 0 0.0018]
         PersistentMemory: false
                   States: Numerator:  [6x1 fi]
                           Denominator:[6x1 fi]


           CoeffWordLength: 16
            CoeffAutoScale: true
                   Signed: true

           InputWordLength: 16
           InputFracLength: 15

          OutputWordLength: 16
                OutputMode: 'AvoidOverflow'

    MultiplicandWordLength: 16
    MultiplicandFracLength: 15

           StateWordLength: 16
            StateAutoScale: true

               ProductMode: 'FullPrecision'

           AccumWordLength: 40
             CastBeforeSum: true

                 RoundMode: 'convergent'
              OverflowMode: 'wrap'

hd.outputwordLength=32

hd =

          FilterStructure: 'Direct-Form I Transposed'
               Arithmetic: 'fixed'
```

```
              Numerator: [1x7 double]
            Denominator: [1 0 0.7777 0 0.1142 0 0.0018]
        PersistentMemory: false
                 States: Numerator:  [6x1 fi]
                         Denominator:[6x1 fi]


          CoeffWordLength: 16
           CoeffAutoScale: true
                   Signed: true

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 32
               OutputMode: 'AvoidOverflow'

    MultiplicandWordLength: 16
    MultiplicandFracLength: 15

          StateWordLength: 16
           StateAutoScale: true

               ProductMode: 'FullPrecision'

          AccumWordLength: 40
             CastBeforeSum: true

                RoundMode: 'convergent'
             OverflowMode: 'wrap'
```

When you create a filter object, this property starts with the value 16.

OverflowMode

The OverflowMode property is specified as one of the following two strings indicating how to respond to overflows in fixed-point arithmetic:

- 'saturate' — saturate overflows.

When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the applicable word length and fraction length properties), these values are quantized to the value of either the largest or smallest representable value, depending on which is closest. `saturate` is the default value for `OverflowMode`.

- `'wrap'` — wrap all overflows to the range of representable values.

  When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the data format properties), these values are wrapped back into that range using modular arithmetic relative to the smallest representable number. You can learn more about modular arithmetic in Fixed-Point Toolbox documentation.

These rules apply to the `OverflowMode` property.

- Applies to the accumulator and output data only.
- Does not apply to coefficients or input data. These always saturate the results.
- Does not apply to products. Products maintain full precision at all times. Your filters do not lose precision in the products.

Note Numbers in floating-point filters that extend beyond the dynamic range overflow to ±inf.

ProductFracLength

After you set `ProductMode` for a fixed-point filter to `SpecifyPrecision`, this property becomes available for you to change. `ProductFracLength` sets the fraction length the filter uses for the results of multiplication operations. Only the FIR filters such as asymmetric FIRs or lattice autoregressive filters include this dynamic property.

Your fraction length can be any negative or positive integer, or zero. In addition, the fraction length you specify can be larger than the associated word length. Generally, the default value is 11 bits.

ProductMode

This property, available when your filter is in fixed-point arithmetic mode, specifies how the filter outputs the results of multiplication operations. All `dfilt` objects include this property when they use fixed-point arithmetic.

When available, you select from one of the following values for `ProductMode`:

- `FullPrecision` — means the filter automatically chooses the word length and fraction length it uses to represent the results of multiplication operations. The setting allow the product to retain the precision provided by the inputs (multiplicands) to the operation.

- `KeepMSB` — means you specify the word length for representing product operation results. The filter sets the fraction length to discard the LSBs, keep the higher order bits in the data, and maintain the precision.

- `KeepLSB` — means you specify the word length for representing the product operation results. The filter sets the fraction length to discard the MSBs, keep the lower order bits, and maintain the precision. Compare to the `KeepMSB` option.

- `SpecifyPrecision` — means you specify the word length and the fraction length to apply to data output from product operations.

When you switch to fixed-point filtering from floating-point, you are most likely going to throw away some data bits after product operations in your filter, perhaps because you have limited resources. When you have to discard some bits, you might choose to discard the least significant bits (LSB) from a result since the resulting quantization error would be small as the LSBs carry less weight. Or you might choose to keep the LSBs because the results have MSBs that are mostly zero, such as when your values are small relative to the range of the format in which they are represented. So the options for `ProductMode` let you choose how to maintain the information you need from the accumulator.

For more information about data formats, word length, and fraction length in fixed-point arithmetic, refer to "Notes About Fraction Length, Word Length, and Precision" on page 8-29.

ProductWordLength

You use `ProductWordLength` to define the data word length used by the output from multiplication operations. Set this property to a value that matches your intended application. For example, the default value is 32 bits, but you can set any word length.

```
set(hq,'arithmetic','fixed');
set(hq,'ProductWordLength',64);
```

Note that `ProductWordLength` applies only to filters whose `Arithmetic` property value is `fixed`.

PersistentMemory

Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter object. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `false` — the filter does not retain memory about filtering operations from one to the next. Maintaining memory (setting `PersistentMemory` to `true`) lets you filter large data sets as collections of smaller subsets and get the same result.

In this example, filter `hd` first filters data `xtot` in one pass. Then you can use `hd` to filter `x` as two separate data sets. The results `ytot` and `ysec` are the same in both cases.

```
xtot=[x,x];
ytot=filter(hd,xtot)
ytot =

         0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
reset(hm1);  % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hd,x) filter(hd,x)]

ysec =

         0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

RoundMode

The `RoundMode` property value specifies the rounding method used for quantizing numerical values. Specify the `RoundMode` property values as one of the following five strings.

RoundMode String	Description of Rounding Algorithm
Ceiling	Round toward positive infinity.
Floor	Round toward negative infinity.
Nearest	Round toward nearest. Ties round toward positive infinity.
Nearest(Convergent)	Round to the closest representable integer. Ties round to the nearest even stored integer. This is the least biased of the methods available in this software.
Round	Round toward nearest. Ties round toward negative infinity for negative numbers, and toward positive infinity for positive numbers.
Zero	Round toward zero.

The choice you make affects only the accumulator and output arithmetic. Coefficient and input arithmetic always round. Finally, products never overflow — they maintain full precision.

ScaleValueFracLength

Filter structures `df1sos`, `df1tsos`, `df2sos`, and `df2tsos` that use `fixed` arithmetic have this property that defines the fraction length applied to the scale values the filter uses between sections. In combination with `CoeffWordLength`, these two properties fully specify how the filter interprets and uses the scale values stored in the property `ScaleValues`. As with fraction length properties, `ScaleValueFracLength` can be any integer,

including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter.

ScaleValues

The `ScaleValues` property values are specified as a scalar (or vector) that introduces scaling for inputs (and the outputs from cascaded sections in the vector case) during filtering:

- When you only have a single section in your filter:
    - Specify the `ScaleValues` property value as a scalar if you only want to scale the input to your filter.
    - Specify the `ScaleValues` property as a vector of length 2 if you want to specify scaling to the input (scaled with the first entry in the vector) and the output (scaled with the last entry in the vector).
- When you have *L* cascaded sections in your filter:
    - Specify the `ScaleValues` property value as a scalar if you only want to scale the input to your filter.
    - Specify the value for the `ScaleValues` property as a vector of length *L*+1 if you want to scale the inputs to every section in your filter, along with the output:

        The first entry of your vector specifies the input scaling

        Each successive entry specifies the scaling at the output of the next section

        The final entry specifies the scaling for the filter output.

The default value for `ScaleValues` is 0.

The interpretation of this property is described as follows with diagrams in "Interpreting the ScaleValues Property" on page 8-84.

Note The value of the `ScaleValues` property is not quantized. Data affected by the presence of a scaling factor in the filter is quantized according to the appropriate data format.

When you apply `normalize` to a fixed-point filter, the value for the `ScaleValues` property is changed accordingly.

It is good practice to choose values for this property that are either positive or negative powers of two.

Interpreting the ScaleValues Property. When you specify the values of the `ScaleValues` property of a quantized filter, the values are entered as a vector, the length of which is determined by the number of cascaded sections in your filter:

- When you have only one section, the value of the `Scalevalues` property can be a scalar or a two-element vector.

- When you have L cascaded sections in your filter, the value of the `ScaleValues` property can be a scalar or an $L+1$-element vector.

The following diagram shows how the `ScaleValues` property values are applied to a quantized filter with only one section.

**Application of ScaleValues
to a Single Section**
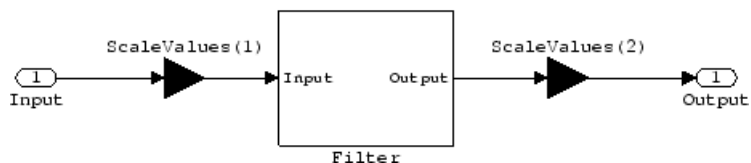
The following diagram shows how the `ScaleValues` property values are applied to a quantized filter with two sections.

**Application of ScaleValues
to Multiple Sections**
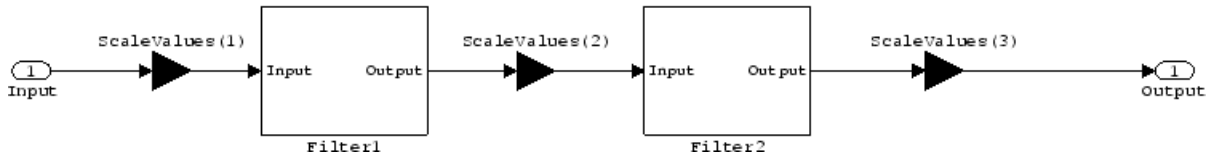
Signed

When you create a dfilt object for fixed-point filtering (you set the property Arithmetic to fixed, the property Signed specifies whether the filter interprets coefficients as signed or unsigned. This setting applies only to the coefficients. While the default setting is true, meaning that all coefficients are assumed to be signed, you can change the setting to false after you create the fixed-point filter.

For example, create a fixed-point direct-form II transposed filter with both negative and positive coefficients, and then change the property value for Signed from true to false to see what happens to the negative coefficient values.

```
hd=dfilt.df2t(-5:5)

hd =

        FilterStructure: 'Direct-Form II Transposed'
              Arithmetic: 'double'
                Numerator: [-5 -4 -3 -2 -1 0 1 2 3 4 5]
              Denominator: 1
         PersistentMemory: false
                   States: [10x1 double]

set(hd,'arithmetic','fixed')
hd.numerator

ans =

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

```
     1     2     3     4     5

set(hd,'signed',false)
hd.numerator

ans =

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

Using unsigned coefficients limits you to using only positive coefficients in your filter. Signed is a dynamic property — you cannot set or change it until you switch the setting for the Arithmetic property to fixed.

SosMatrix

When you convert a dfilt object to second-order section form, or create a second-order section filter, sosMatrix holds the filter coefficients as property values. Using the double data type by default, the matrix is in [sections coefficients per section] form, displayed as [15-x-6] for filters with 6 coefficients per section and 15 sections, [15 6].

To demonstrate, the following code creates an order 30 filter using second-order sections in the direct-form II transposed configuration. Notice the sosMatrix property contains the coefficients for all the sections.

```
d = fdesign.lowpass('n,fc',30,0.5);
hd = butter(d);

hd =

  FilterStructure: 'Direct-Form II, Second-Order Sections'
       Arithmetic: 'double'
         sosMatrix: [15x6 double]
      ScaleValues: [16x1 double]
PersistentMemory: false
           States: [2x15 double]

hd.arithmetic='fixed'

hd =
```

```
       FilterStructure: 'Direct-Form II, Second-Order Sections'
            Arithmetic: 'fixed'
              sosMatrix: [15x6 double]
           ScaleValues: [16x1 double]
      PersistentMemory: false
                 States: [1x1 embedded.fi]

       CoeffWordLength: 16
        CoeffAutoScale: true
                Signed: true

       InputWordLength: 16
       InputFracLength: 15

        SectionInputWordLength: 16
         SectionInputAutoScale: true

      SectionOutputWordLength: 16
       SectionOutputAutoScale: true

             OutputWordLength: 16
                   OutputMode: 'AvoidOverflow'

              StateWordLength: 16
              StateFracLength: 15

                  ProductMode: 'FullPrecision'

              AccumWordLength: 40
                CastBeforeSum: true

                    RoundMode: 'convergent'
                 OverflowMode: 'wrap'

hd.sosMatrix

ans =

    1.0000    2.0000    1.0000    1.0000         0    0.9005
```

1.0000	2.0000	1.0000	1.0000	0	0.7294
1.0000	2.0000	1.0000	1.0000	0	0.5888
1.0000	2.0000	1.0000	1.0000	0	0.4724
1.0000	2.0000	1.0000	1.0000	0	0.3755
1.0000	2.0000	1.0000	1.0000	0	0.2948
1.0000	2.0000	1.0000	1.0000	0	0.2275
1.0000	2.0000	1.0000	1.0000	0	0.1716
1.0000	2.0000	1.0000	1.0000	0	0.1254
1.0000	2.0000	1.0000	1.0000	0	0.0878
1.0000	2.0000	1.0000	1.0000	0	0.0576
1.0000	2.0000	1.0000	1.0000	0	0.0344
1.0000	2.0000	1.0000	1.0000	0	0.0173
1.0000	2.0000	1.0000	1.0000	0	0.0062
1.0000	2.0000	1.0000	1.0000	0	0.0007

The SOS matrix is an M-by-6 matrix, where M is the number of sections in the second-order section filter. Filter hd has M equal to 15 as shown above (15 rows). Each row of the SOS matrix contains the numerator and denominator coefficients (b's and a's) and the scale factors of the corresponding section in the filter.

SectionInputAutoScale

Second-order section filters include this property that determines who the filter handles data in the transitions from one section to the next in the filter.

How the filter represents the data passing from one section to the next depends on the property value of SectionInputAutoScale. The representation the filter uses between the filter sections depends on whether the value of SectionInputAutoScale is true or false.

- SectionInputAutoScale = true means the filter chooses the fraction length to maintain the value of the data between sections as close to the output values from the previous section as possible. true is the default setting.

- SectionInputAutoScale = false removes the automatic scaling of the fraction length for the intersection data and exposes the property that controls the coefficient fraction length (SectionInputFracLength) so you can change it. For example, if the filter is a second-order, direct form FIR filter, setting SectionInputAutoScale to false exposes the

SectionInputFracLength property that specifies the fraction length applied to data between the sections.

SectionInputFracLength

Second-order section filters use quantizers at the input to each section of the filter. The quantizers apply to the input data entering each filter section. Note that the quantizers for each section are the same. To set the fraction length for interpreting the input values, use the property value in SectionInputFracLength.

In combination with CoeffWordLength, SectionInputFracLength fully determines how the filter interprets and uses the state values stored in the property States. As with all word and fraction length properties, SectionInputFracLength can be any integer, including integers larger than CoeffWordLength, and positive or negative integers. 15 bits is the default value when you create the filter object.

SectionInputWordLength

SOS filters are composed of sections, each one a second-order filter. Filtering data input to the filter involves passing the data through each filter section. SectionInputWordLength specifies the word length applied to data as it enters one filter section from the previous section. Only second-order implementations of direct-form I transposed and direct-form II transposed filters include this property.

The following diagram shows an SOS filter composed of sections (the bottom part of the diagram) and a possible internal structure of each Section (the top portion of the diagram), in this case — a direct form I transposed second order sections filter structure. Note that the output of each section is fed through a multiplier. If the gain of the multiplier =1, then the last Cast block of the Section is ignored, and the format of the output is NumSumQ.

* : only if CastBeforeSum is True
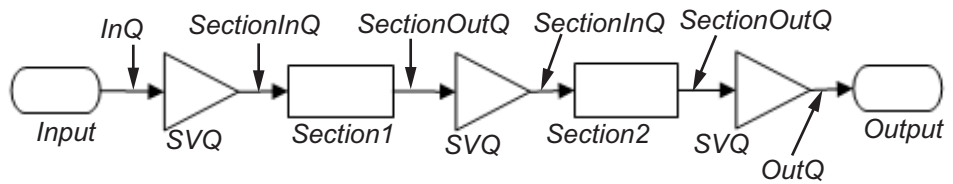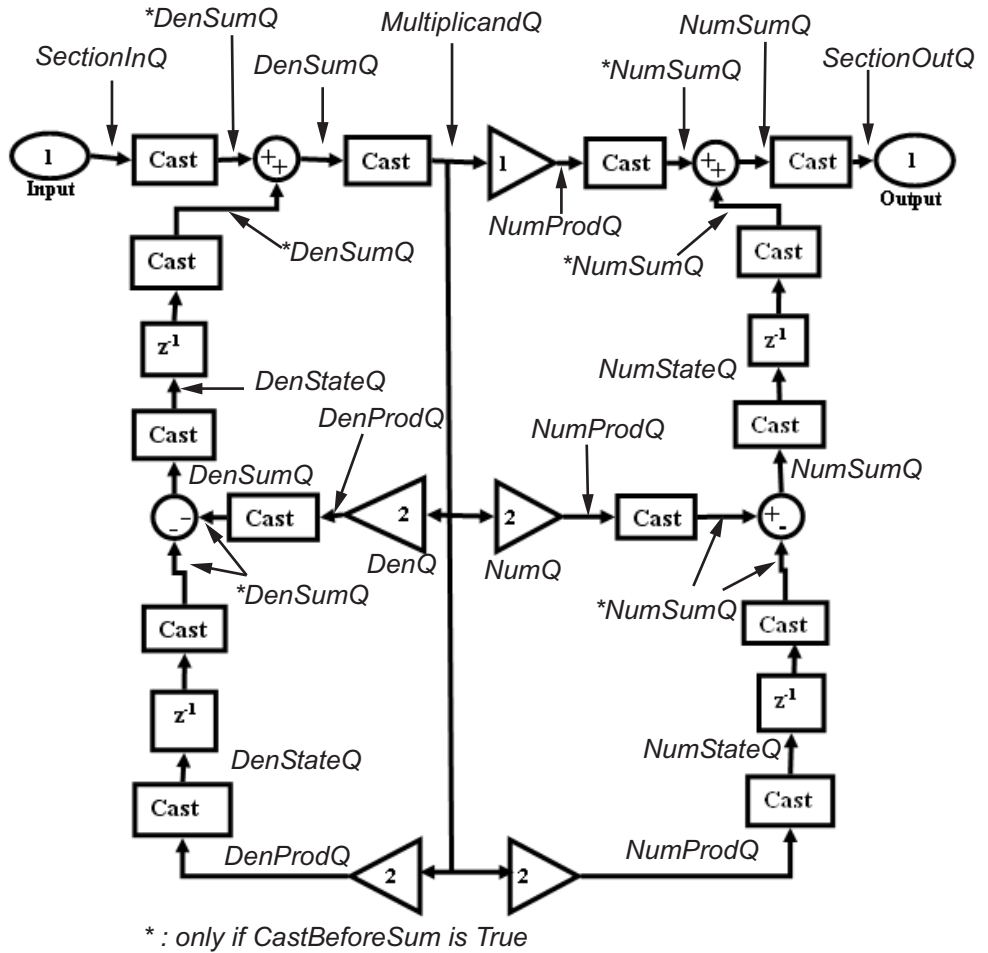
`SectionInputWordLength` defaults to 16 bits.

SectionOutputAutoScale

Second-order section filters include this property that determines who the filter handles data in the transitions from one section to the next in the filter.

How the filter represents the data passing from one section to the next depends on the property value of `SectionOutputAutoScale`. The representation the filter uses between the filter sections depends on whether the value of `SectionOutputAutoScale` is `true` or `false`.

- `SectionOutputAutoScale` = `true` means the filter chooses the fraction length to maintain the value of the data between sections as close to the output values from the previous section as possible. `true` is the default setting.

- `SectionOutputAutoScale` = `false` removes the automatic scaling of the fraction length for the intersection data and exposes the property that controls the coefficient fraction length (`SectionOutputFracLength`) so you can change it. For example, if the filter is a second-order, direct form FIR filter, setting `SectionOutputAutoScale` = `false` exposes the `SectionOutputFracLength` property that specifies the fraction length applied to data between the sections.

SectionOutputFracLength

Second-order section filters use quantizers at the output from each section of the filter. The quantizers apply to the output data leaving each filter section. Note that the quantizers for each section are the same. To set the fraction length for interpreting the output values, use the property value in `SectionOutputFracLength`.

In combination with `CoeffWordLength`, `SectionOutputFracLength` determines how the filter interprets and uses the state values stored in the property `States`. As with all fraction length properties, `SectionOutputFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

SectionOutputWordLength

SOS filters are composed of sections, each one a second-order filter. Filtering data input to the filter involves passing the data through each filter section. `SectionOutputWordLength` specifies the word length applied to data as it leaves one filter section to go to the next. Only second-order implementations direct-form I transposed and direct-form II transposed filters include this property.

The following diagram shows an SOS filter composed of sections (the bottom part of the diagram) and a possible internal structure of each Section (the top portion of the diagram), in this case — a direct form I transposed second order sections filter structure. Note that the output of each section is fed through a multiplier. If the gain of the multiplier =1, then the last Cast block of the Section is ignored, and the format of the output is NumSumQ.
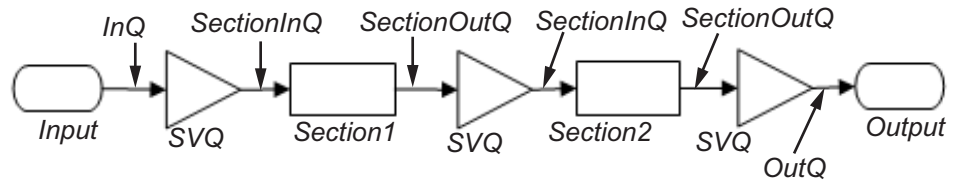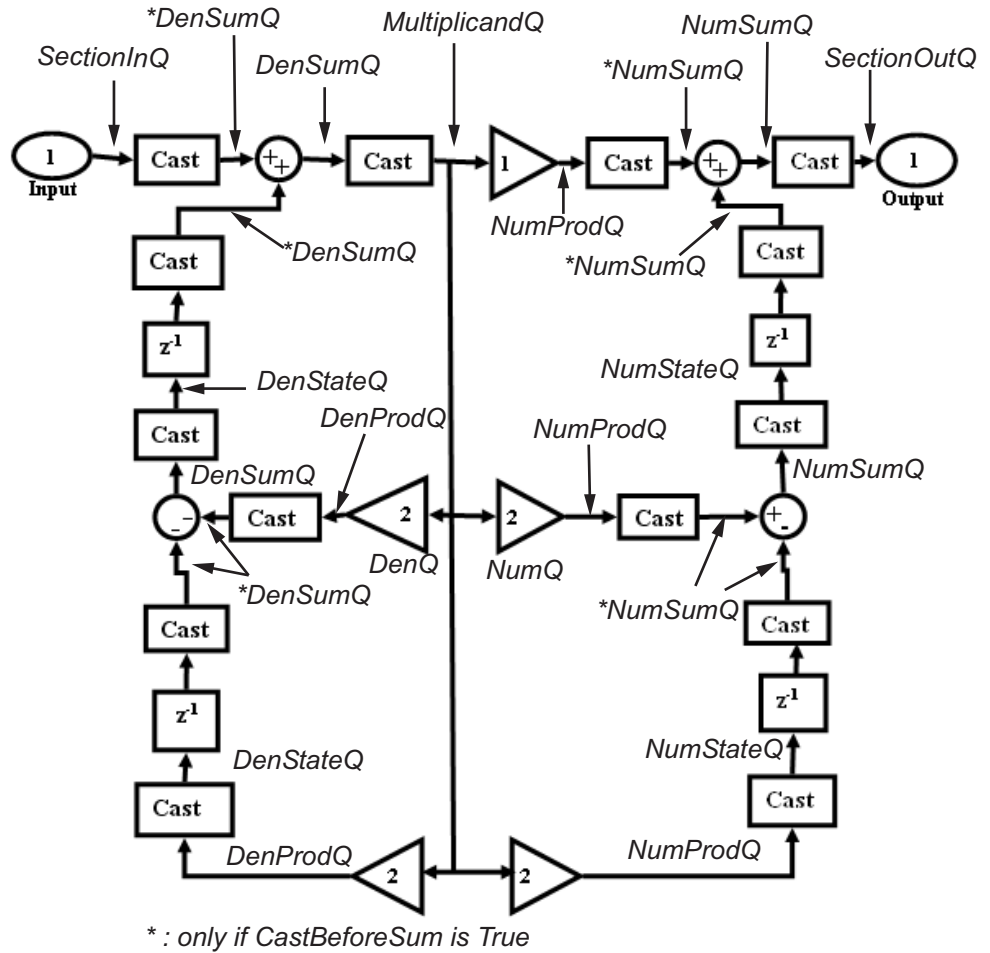
SectionOutputWordLength defaults to 16 bits.

StateAutoScale

Although all filters use states, some do not allow you to choose whether the filter automatically scales the state values to prevent overruns or bad arithmetic errors. You select either of the following settings:

- StateAutoScale = true means the filter chooses the fraction length to maintain the value of the states as close to the double-precision values as possible. When you change the word length applied to the states (where allowed by the filter structure), the filter object changes the fraction length to try to accommodate the change. true is the default setting.

- StateAutoScale = false removes the automatic scaling of the fraction length for the states and exposes the property that controls the coefficient fraction length so you can change it. For example, in a direct form I transposed SOS FIR filter, setting StateAutoScale = false exposes the NumStateFracLength and DenStateFracLength properties that specify the fraction length applied to states.

Each of the following filter structures provides the StateAutoScale property:

- df1t
- df1tsos
- df2t
- df2tsos
- dffirt

Other filter structures do not include this property.

StateFracLength

Filter states stored in the property States have both word length and fraction length. To set the fraction length for interpreting the stored filter object state values, use the property value in StateFracLength.

In combination with `CoeffWordLength`, `StateFracLength` fully determines how the filter interprets and uses the state values stored in the property `States`.

As with all fraction length properties, `StateFracLength` can be any integer, including integers larger than `CoeffWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

States

Digital filters are dynamic systems. The behavior of dynamic systems (their response) depends on the input (stimulus) to the system and the current or previous *state* of the system. You can say the system has memory or inertia. All fixed- or floating-point digital filters (as well as analog filters) have states.

Filters use the states to compute the filter output for each input sample, as well using them while filtering in loops to maintain the filter state between loop iterations. This toolbox assumes zero-valued initial conditions (the dynamic system is at rest) by default when you filter the first input sample. Assuming the states are zero initially does not mean the states are not used; they are, but arithmetically they do not have any effect.

Filter objects store the state values in the property `States`. The number of stored states depends on the filter implementation, since the states represent the delays in the filter implementation.

When you review the display for a filter object with fixed arithmetic, notice that the states return an embedded `fi` object, as you see here.

```
b = ellip(6,3,50,300/500);
hd=dfilt.dffir(b)


hd =

       FilterStructure: 'Direct-Form FIR'
             Arithmetic: 'double'
              Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858 0.2938 0.0773]
         PersistentMemory: false
                 States: [6x1 double]
```

```
hd.arithmetic='fixed'

hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858 0.2938 0.0773]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

          CoeffWordLength: 16
           CoeffAutoScale: 'on'
                   Signed: 'on'

          InputWordLength: 16
           InputFracLength: 15

         OutputWordLength: 16
               OutputMode: 'AvoidOverflow'

              ProductMode: 'FullPrecision'

          AccumWordLength: 40
             CastBeforeSum: 'on'

                RoundMode: 'convergent'
             OverflowMode: 'wrap'

          InheritSettings: 'off'
```

fi objects provide fixed-point support for the filters. To learn more about the details about fi objects, refer to your Fixed-Point Toolbox documentation.

The property States lets you use a fi object to define how the filter interprets the filter states. For example, you can create a fi object in MATLAB, then assign the object to States, as follows:

```
statefi=fi([],16,12)
```

```
statefi =

[]
    DataTypeMode = Fixed-point: binary point scaling
           Signed = true
      Wordlength = 16
Fractionlength = 12
```

This `fi` object does not have a value associated (notice the `[]` input argument to `fi` for the value), and it has word length of 16 bits and fraction length of 12 bit. Now you can apply `statefi` to the `States` property of the filter `hd`.

```
set(hd,'States',statefi);
Warning: The 'States' property will be reset to the value
specified at construction before filtering.
Set the 'PersistentMemory' flag to 'True'
to avoid changing this property value.
hd

hd =

         FilterStructure: 'Direct-Form FIR'
              Arithmetic: 'fixed'
               Numerator: [0.0773 0.2938 0.5858 0.7239 0.5858
                           0.2938 0.0773]
        PersistentMemory: false
                  States: [1x1 embedded.fi]

         CoeffWordLength: 16
          CoeffAutoScale: 'on'
                  Signed: 'on'

         InputWordLength: 16
         InputFracLength: 15

        OutputWordLength: 16
              OutputMode: 'AvoidOverflow'

             ProductMode: 'FullPrecision'
         AccumWordLength: 40
```

```
            CastBeforeSum: 'on'

                RoundMode: 'convergent'
             OverflowMode: 'wrap'
```

StateWordLength

While all filters use states, some do not allow you to directly change the state representation — the word length and fraction lengths — independently. For the others, `StateWordLength` specifies the word length, in bits, the filter uses to represent the states. Filters that do not provide direct state word length control include:

- `df1`

- `dfasymfir`

- `dffir`

- `dfsymfir`

For these structures, the filter derives the state format from the input format you choose for the filter — except for the `df1` IIR filter. In this case, the numerator state format comes from the input format and the denominator state format comes from the output format. All other filter structures provide control of the state format directly.

TapSumFracLength

Direct-form FIR filter objects, both symmetric and antisymmetric, use this property. To set the fraction length for output from the sum operations that involve the filter tap weights, use the property value in `TapSumFracLength`. To enable this property, set the `TapSumMode` to `SpecifyPrecision` in your filter.

As you can see in this code example that creates a fixed-point asymmetric FIR filter, the `TapSumFracLength` property becomes available after you change the `TapSumMode` property value.

```
hd=dfilt.dfasymfir

hd =
```

```
              FilterStructure: 'Direct-Form Antisymmetric FIR'
                   Arithmetic: 'double'
                    Numerator: 1
             PersistentMemory: false
                       States: [0x1 double]

set(hd,'arithmetic','fixed');
hd

hd =

              FilterStructure: 'Direct-Form Antisymmetric FIR'
                   Arithmetic: 'fixed'
                    Numerator: 1
             PersistentMemory: false
                       States: [1x1 embedded.fi]

               CoeffWordLength: 16
                CoeffAutoScale: true
                       Signed: true

               InputWordLength: 16
               InputFracLength: 15

              OutputWordLength: 16
                    OutputMode: 'AvoidOverflow'

                    TapSumMode: 'KeepMSB'
              TapSumWordLength: 17

                   ProductMode: 'FullPrecision'

               AccumWordLength: 40

                 CastBeforeSum: true
                     RoundMode: 'convergent'
                  OverflowMode: 'wrap'
```

With the filter now in fixed-point mode, you can change the `TapSumMode` property value to `SpecifyPrecision`, which gives you access to the `TapSumFracLength` property.

```
set(hd,'TapSumMode','SpecifyPrecision');
hd

hd =

          FilterStructure: 'Direct-Form Antisymmetric FIR'
               Arithmetic: 'fixed'
                Numerator: 1
         PersistentMemory: false
                   States: [1x1 embedded.fi]

          CoeffWordLength: 16
            CoeffAutoScale: true
                   Signed: true

          InputWordLength: 16
          InputFracLength: 15

         OutputWordLength: 16
               OutputMode: 'AvoidOverflow'

               TapSumMode: 'SpecifyPrecision'
         TapSumWordLength: 17
         TapSumFracLength: 15

              ProductMode: 'FullPrecision'

          AccumWordLength: 40

            CastBeforeSum: true
                RoundMode: 'convergent'
             OverflowMode: 'wrap'
```

In combination with `TapSumWordLength`, `TapSumFracLength` fully determines how the filter interprets and uses the state values stored in the property `States`.

As with all fraction length properties, `TapSumFracLength` can be any integer, including integers larger than `TapSumWordLength`, and positive or negative integers. 15 bits is the default value when you create the filter object.

TapSumMode

This property, available only after your filter is in fixed-point mode, specifies how the filter outputs the results of summation operations that involve the filter tap weights. Only symmetric (`dfilt.dfsymfir`) and antisymmetric (`dfilt.dfasymfir`) FIR filters use this property.

When available, you select from one of the following values:

- `FullPrecision` — means the filter automatically chooses the word length and fraction length to represent the results of the sum operation so they retain all of the precision provided by the inputs (addends).

- `KeepMSB` — means you specify the word length for representing tap sum summation results to keep the higher order bits in the data. The filter sets the fraction length to discard the LSBs from the sum operation. This is the default property value.

- `KeepLSB` — means you specify the word length for representing tap sum summation results to keep the lower order bits in the data. The filter sets the fraction length to discard the MSBs from the sum operation. Compare to the `KeepMSB` option.

- `SpecifyPrecision` — means you specify the word and fraction lengths to apply to data output from the tap sum operations.

TapSumWordLength

Specifies the word length the filter uses to represent the output from tap sum operations. The default value is 17 bits. Only `dfasymfir` and `dfsymfir` filters include this property.

Adaptive Filter Properties

In this section...

Property Summaries

The following table summarizes the adaptive filter properties and provides a brief description of each. Full descriptions of each property, in alphabetical order, are given in the subsequent section.

Property	Description
Algorithm	Reports the algorithm the object uses for adaptation. When you construct your adaptive filter object, this property is set automatically by the constructor, such as adaptfilt.nlms creating an adaptive filter that uses the normalized LMS algorithm. You cannot change the value — it is read only.
AvgFactor	Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. AvgFactor should lie between zero and one. For default filter objects, AvgFactor equals (1 - step). lambda is the input argument that represents AvgFactor
BkwdPredErrorPower	Returns the minimum mean-squared prediction error. Refer to [3] in the bibliography for details about linear prediction.
BkwdPrediction	Returns the predicted samples generated during adaptation. Refer to [3] in the bibliography for details about linear prediction.

Property	Description
Blocklength	Block length for the coefficient updates. This must be a positive integer such that (1/blocklength) is also an integer. For faster execution, blocklength should be a power of two. blocklength defaults to two.
Coefficients	Vector containing the initial filter coefficients. It must be a length l vector where l is the number of filter coefficients. coeffs defaults to length l vector of zeros when you do not provide the argument for input.
ConversionFactor	Conversion factor defaults to the matrix [1 -1] that specifies soft-constrained initialization. This is the gamma input argument for some of the fast transversal algorithms.
Delay	Update delay given in time samples. This scalar should be a positive integer—negative delays do not work. delay defaults to 1 for most algorithms.
DesiredSignalStates	Desired signal states of the adaptive filter. dstates defaults to a zero vector with length equal to (blocklen - 1) or (swblocklen - 1) depending on the algorithm.
EpsilonStates	Vector of the epsilon values of the adaptive filter. EpsilonStates defaults to a vector of zeros with (projectord - 1) elements.
ErrorStates	Vector of the adaptive filter error states. ErrorStates defaults to a zero vector with length equal to (projectord - 1).
FFTCoefficients	Stores the discrete Fourier transform of the filter coefficients in coeffs.
FFTStates	Stores the states of the FFT of the filter coefficients during adaptation.
FilteredInputStates	Vector of filtered input states with length equal to l - 1.

Property	Description
FilterLength	Contains the length of the filter. Note that this is not the filter order. Filter length is 1 greater than filter order. Thus a filter with length equal to 10 has filter order equal to 9.
ForgettingFactor	Determines how the RLS adaptive filter uses past data in each iteration. You use the forgetting factor to specify whether old data carries the same weight in the algorithm as more recent data.
FwdPredErrorPower	Returns the minimum mean-squared prediction error in the forward direction. Refer to [3] in the bibliography for details about linear prediction.
FwdPrediction	Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.
InitFactor	Soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. Called delta as an input argument, this defaults to one.
InvCov	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix. Dimensions are l-by-l, where l is the filter length.
KalmanGain	Empty when you construct the object, this gets populated after you run the filter.
KalmanGainStates	Contains the states of the Kalman gain updates during adaptation.

Property	Description
Leakage	Contains the setting for leakage in the adaptive filter algorithm. Using a leakage factor that is not 1 forces the weights to adapt even when they have found the minimum error solution. Forcing the adaptation can improve the numerical performance of the LMS algorithm.
OffsetCov	Contains the offset covariance matrix.
Offset	Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors.
Power	A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.
ProjectionOrder	Projection order of the affine projection algorithm. projectord defines the size of the input signal covariance matrix and defaults to two.
ReflectionCoeffs	Coefficients determined for the reflection portion of the filter during adaptation.
ReflectionCoeffsStep	Size of the steps used to determine the reflection coefficients.
PersistentMemory	Specifies whether to reset the filter states and memory before each filtering operation. Lets you decide whether your filter retains states and coefficients from previous filtering runs.
SecondaryPathCoeffs	A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor.
SecondaryPathEstimate	An estimate of the secondary path filter model.

Property	Description
SecondaryPathStates	The states of the secondary path filter, the unknown system.
SqrtCov	Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.
SqrtlnvCov	Square root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.
States	Vector of the adaptive filter states. states defaults to a vector of zeros whose length depends on the chosen algorithm. Usually the length is a function of the filter length l and another input argument to the filter object, such as projectord.
StepSize	Reports the size of the step taken between iterations of the adaptive filter process. Each adaptfilt object has a default value that best meets the needs of the algorithm.
SwBlockLength	Block length of the sliding window. This integer must be at least as large as the filter length. swblocklen defaults to 16.

Like dfilt objects, adaptfilt objects have properties that govern their behavior and store some of the results of filtering operations. The following pages list, in alphabetical order, the name of every property associated with adaptfilt objects. Note that not all adaptfilt objects have all of these properties. To view the properties of a particular adaptive filter, such as an adaptfilt.bap filter, use get with the object handle, like this:

```
ha = adaptfilt.bap(32,0.5,4,1.0);
get(ha)
        PersistentMemory: false
               Algorithm: 'Block Affine Projection FIR Adaptive Filter'
              FilterLength: 32
              Coefficients: [1x32 double]
```

```
         States: [35x1 double]
       StepSize: 0.5000
ProjectionOrder: 4
      OffsetCov: [4x4 double]
```

get shows you the properties for ha and the values for the properties. Entering the object handle returns the same values and properties without the formatting of the list and the more familiar property names.

Property Details for Adaptive Filter Properties

Algorithm

Reports the algorithm the object uses for adaptation. When you construct you adaptive filter object, this property is set automatically. You cannot change the value—it is read only.

AvgFactor

Averaging factor used to compute the exponentially-windowed estimates of the powers in the transformed signal bins for the coefficient updates. AvgFactor should lie between zero and one. For default filter objects, AvgFactor equals (1 - step). lambda is the input argument that represent AvgFactor

BkwdPredErrorPower

Returns the minimum mean-squared prediction error in the backward direction. Refer to [3] in the bibliography for details about linear prediction.

BkwdPrediction

When you use an adaptive filter that does backward prediction, such as adaptfilt.ftf, one property of the filter contains the backward prediction coefficients for the adapted filter. With these coefficient, the forward coefficients, and the system under test, you have the full set of knowledge of how the adaptation occurred. Two values stored in properties compose the BkwdPrediction property:

• Coefficients, which contains the coefficients of the system under test, as determined using backward predictions process.

- Error, which is the difference between the filter coefficients determined by backward prediction and the actual coefficients of the sample filter. In this example, `adaptfilt.ftf` identifies the coefficients of an unknown FIR system.

```
x   = randn(1,500);     % Input to the filter
b   = fir1(31,0.5);     % FIR system to be identified
n   = 0.1*randn(1,500); % Observation noise signal
d   = filter(b,1,x)+n;  % Desired signal
N   = 31;               % Adaptive filter order
lam = 0.99;             % RLS forgetting factor
del = 0.1;              % Soft-constrained initialization factor
ha = adaptfilt.ftf(32,lam,del);
[y,e] = filter(ha,x,d);

ha

ha =

               Algorithm: 'Fast Transversal Least-Squares Adaptive Filter'
            FilterLength: 32
            Coefficients: [1x32 double]
                  States: [31x1 double]
         ForgettingFactor: 0.9900
               InitFactor: 0.1000
             FwdPrediction: [1x1 struct]
            BkwdPrediction: [1x1 struct]
                KalmanGain: [32x1 double]
          ConversionFactor: 0.7338
          KalmanGainStates: [32x1 double]
           PersistentMemory: false

ha.coefficients

ans =

  Columns 1 through 8

    -0.0055    0.0048    0.0045    0.0146   -0.0009    0.0002   -0.0019    0.0008
```

```
  Columns 9 through 16

   -0.0142   -0.0226    0.0234    0.0421   -0.0571   -0.0807    0.1434    0.4620

  Columns 17 through 24

    0.4564    0.1532   -0.0879   -0.0501    0.0331    0.0361   -0.0266   -0.0220

  Columns 25 through 32

    0.0231    0.0026   -0.0063   -0.0079    0.0032    0.0082    0.0033    0.0065

ha.bkwdprediction

ans =

    Coeffs: [1x32 double]
     Error: 82.3394

>> ha.bkwdprediction.coeffs

ans =

  Columns 1 through 8

    0.0067    0.0186    0.1114   -0.0150   -0.0239   -0.0610   -0.1120   -0.1026

  Columns 9 through 16

    0.0093   -0.0399   -0.0045    0.0622    0.0997    0.0778    0.0646   -0.0564

  Columns 17 through 24

    0.0775    0.0814    0.0057    0.0078    0.1271   -0.0576    0.0037   -0.0200

  Columns 25 through 32

   -0.0246    0.0180   -0.0033    0.1222    0.0302   -0.0197   -0.1162    0.0285
```

Blocklength

Block length for the coefficient updates. This must be a positive integer such that (1/blocklen) is also an integer. For faster execution, `blocklen` should be a power of two. `blocklen` defaults to two.

Coefficients

Vector containing the initial filter coefficients. It must be a length `l` vector where `l` is the number of filter coefficients. `coeffs` defaults to length `l` vector of zeros when you do not provide the argument for input.

ConversionFactor

Conversion factor defaults to the matrix [1 -1] that specifies soft-constrained initialization. This is the `gamma` input argument for some of the fast transversal algorithms.

Delay

Update delay given in time samples. This scalar should be a positive integer — negative delays do not work. `delay` defaults to 1 for most algorithms.

DesiredSignalStates

Desired signal states of the adaptive filter. `dstates` defaults to a zero vector with length equal to (`blocklen` - 1) or (`swblocklen` - 1) depending on the algorithm.

EpsilonStates

Vector of the epsilon values of the adaptive filter. `EpsilonStates` defaults to a vector of zeros with (`projectord` - 1) elements.

ErrorStates

Vector of the adaptive filter error states. `ErrorStates` defaults to a zero vector with length equal to (`projectord` - 1).

FFTCoefficients

Stores the discrete Fourier transform of the filter coefficients in `coeffs`.

FFTStates

Stores the states of the FFT of the filter coefficients during adaptation.

FilteredInputStates

Vector of filtered input states with length equal to l - 1.

FilterLength

Contains the length of the filter. Note that this is not the filter order. Filter length is 1 greater than filter order. Thus a filter with length equal to 10 has filter order equal to 9.

ForgettingFactor

Determines how the RLS adaptive filter uses past data in each iteration. You use the forgetting factor to specify whether old data carries the same weight in the algorithm as more recent data.

This is a scalar and should lie in the range (0, 1]. It defaults to 1. Setting forgetting factor = 1 denotes infinite memory while adapting to find the new filter. Note that this is the lambda input argument.

FwdPredErrorPower

Returns the minimum mean-squared prediction error in the forward direction. Refer to [3] in the bibliography for details about linear prediction.

FwdPrediction

Contains the predicted values for samples during adaptation. Compare these to the actual samples to get the error and power.

InitFactor

Returns the soft-constrained initialization factor. This scalar should be positive and sufficiently large to prevent an excessive number of Kalman gain rescues. delta defaults to one.

InvCov

Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix. Dimensions are l-by-l, where l is the filter length.

KalmanGain

Empty when you construct the object, this gets populated after you run the filter.

KalmanGainStates

Contains the states of the Kalman gain updates during adaptation.

Leakage

Contains the setting for leakage in the adaptive filter algorithm. Using a leakage factor that is not 1 forces the weights to adapt even when they have found the minimum error solution. Forcing the adaptation can improve the numerical performance of the LMS algorithm.

OffsetCov

Contains the offset covariance matrix.

Offset

Specifies an optional offset for the denominator of the step size normalization term. You must specify offset to be a scalar greater than or equal to zero. Nonzero offsets can help avoid a divide-by-near-zero condition that causes errors.

Use this to avoid dividing by zero or by very small numbers when input signal amplitude becomes very small, or dividing by very small numbers when any of the FFT input signal powers become very small. offset defaults to one.

Power

A vector of 2*l elements, each initialized with the value delta from the input arguments. As you filter data, Power gets updated by the filter process.

ProjectionOrder

Projection order of the affine projection algorithm. `projectord` defines the size of the input signal covariance matrix and defaults to two.

ReflectionCoeffs

For adaptive filters that use reflection coefficients, this property stores them.

ReflectionCoeffsStep

As the adaptive filter changes coefficient values during adaptation, the step size used between runs is stored here.

PersistentMemory

Determines whether the filter states and coefficients get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter.

`PersistentMemory` returns to zero any property value that the filter changes during processing. Property values that the filter does not change are not affected. Defaults to `false`.

SecondaryPathCoeffs

A vector that contains the coefficient values of your secondary path from the output actuator to the error sensor.

SecondaryPathEstimate

An estimate of the secondary path filter model.

SecondaryPathStates

The states of the secondary path filter, the unknown system.

SqrtCov

Upper-triangular Cholesky (square root) factor of the input covariance matrix. Initialize this matrix with a positive definite upper triangular matrix.

SqrtInvCov

Square root of the inverse of the sliding window input signal covariance matrix. This square matrix should be full-ranked.

States

Vector of the adaptive filter states. `states` defaults to a vector of zeros whose length depends on the chosen algorithm. Usually the length is a function of the filter length `l` and another input argument to the filter object, such as projectord.

StepSize

Reports the size of the step taken between iterations of the adaptive filter process. Each `adaptfilt` object has a default value that best meets the needs of the algorithm.

SwBlockLength

Block length of the sliding window. This integer must be at least as large as the filter length. `swblocklength` defaults to `16`.

References for Adaptive Filters

[1] Griffiths, L.J., *A Continuously Adaptive Filter Implemented as a Lattice Structure*, Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, Hartford, CT, pp. 683-686, 1977.

[2] Hayes, M.H., *Statistical Digital Signal Processing and Modeling*, John Wiley and Sons, 1996.

[3] Haykin, S., *Adaptive Filter Theory*, Third Edition, Prentice-Hall, Inc., 1996.

Multirate Filter Properties

In this section...
"Property Summaries" on page 8-115
"Property Details for Multirate Filter Properties" on page 8-120
"References for Multirate Filters" on page 8-130

Property Summaries

The following table summarizes the multirate filter properties and provides a brief description of each. Full descriptions of each property are given in the subsequent section.

Name	Values	Default	Description
BlockLength	Positive integers	100	Length of each block of data input to the FFT used in the filtering. fftfirinterp multirate filters include this property.
DecimationFactor	Any positive integer	2	Amount to reduce the input sampling rate.
DifferentialDelay	Any integer	1	Sets the differential delay for the filter. Usually a value of one or two is appropriate.

Name	Values	Default	Description
FilterInternals	FullPrecision, MinWordlengths, SpecifyWordLengths, SpecifyPrecision	FullPrecision	Controls whether the filter sets the output word and fraction lengths, and the accumulator word and fraction lengths automatically to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. SpecifyPrecision exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.
FilterStructure	mfilt structure string	None	Describes the signal flow for the filter object, including all of the active elements that perform operations during filtering — gains, delays, sums, products, and input/output. You cannot set this property — it is always read only and results from your choice of mfilt object.

Name	Values	Default	Description
InputOffset	Integers	0	Contains the number of input data samples processed without generating an output sample. InputOffset = mod(length(nx),m) where nx is the number of input samples that have been processed so far and m is the decimation factor.
InterpolationFactor	Positive integers	2	Interpolation factor for the filter. 1 specifies the amount to increase the input sampling rate.
NumberOfSections	Any positive integer	2	Number of sections used in the decimator, or in the comb and integrator portions of CIC filters.
Numerator	Array of double values	No default values	Vector containing the coefficients of the FIR lowpass filter used for interpolation.
OverflowMode	saturate, [wrap]	wrap	Sets the mode used to respond to overflow conditions in fixed-point arithmetic. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic. The choice you make affects only the accumulator and output

Name	Values	Default	Description
			arithmetic. Coefficient and input arithmetic always saturates. Finally, products never overflow — they maintain full precision.
PolyphaseAccum	Values depend on filter type. Either double, single, or fixed-point object	0	Stores the value remaining in the accumulator after the filter processes the last input sample. The stored value for `PolyphaseAccum` affects the next output when `PersistentMemory` is `true` and `InputOffset` is not equal to 0. Always provides full precision values. Compare the `AccumWordLength` and `AccumFracLength`.
PersistentMemory	`false` or `true`	`false`	Determines whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.

Name	Values	Default	Description
RateChangeFactors	[l,m]	[2,3] or [3,2]	Reports the decimation (m) and interpolation (l) factors for the filter object. Combining these factors results in the final rate change for the signal. The default changes depending on whether the filter decimates or interpolates.
States	Any m+1-by-n matrix of double values	2-by-2 matrix, int32	Stored conditions for the filter, including values for the integrator and comb sections. n is the number of filter sections and m is the differential delay. Stored in a filtstates object.
SpecifyWordLengths	Vector of integers	[16 16 16 16] bits	
WordLengthPerSection	Any integer or a vector of length 2*n	16	Defines the word length used in each section while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using 'wrap' arithmetic). Enter WordLengthPerSection as a scalar or vector of length 2*n, where n is the number of sections. When WordLengthPerSection is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator.

The following sections provide details about the properties that govern the way multirate filter work. Creating any multirate filter object puts in place a number of these properties. The following pages list the `mfilt` object properties in alphabetical order.

Property Details for Multirate Filter Properties

BitsPerSection
Any integer or a vector of length 2*n.

Defines the bits per section used while accumulating the data in the integrator sections or while subtracting the data during the comb sections (using `wrap` arithmetic). Enter `bps` as a scalar or vector of length 2*n, where `n` is the number of sections. When `bps` is a scalar, the scalar value is applied to each filter section. The default is 16 for each section in the decimator.

BlockLength
Length of each block of input data used in the filtering.

`mfilt.fftfirinterp` objects process data in blocks whose length is determined by the value you set for the `BlockLength` property. By default the property value is 100. When you set the `BlockLength` value, try choosing a value so that [`BlockLength` + `length`(filter order)] is a power of two.

Larger block lengths generally reduce the computation time.

DecimationFactor
Decimation factor for the filter. `m` specifies the amount to reduce the sampling rate of the input signal. It must be an integer. You can enter any integer value. The default value is 2.

DifferentialDelay
Sets the differential delay for the filter. Usually a value of one or two is appropriate. While you can set any value, the default is one and the maximum is usually two.

FilterInternals

Similar to the FilterInternals pane in FDATool, this property controls whether the filter sets the output word and fraction lengths automatically, and the accumulator word and fraction lengths automatically as well, to maintain the best precision results during filtering. The default value, FullPrecision, sets automatic word and fraction length determination by the filter. Setting FilterInternals to SpecifyPrecision exposes the output and accumulator related properties so you can set your own word and fraction lengths for them.

About FilterInternals Mode. There are four usage modes for this that you set using the FilterInternals property in multirate filters.

- FullPrecision — All word and fraction lengths set to B_{max} + 1, called B_{accum} by Fred Harris in [2]. Full precision is the default setting.

- MinWordLengths — Minimum Word Lengths

- SpecifyWordLengths — Specify Word Lengths

- SpecifyPrecision — Specify Precision

Full Precision

In full precision mode, the word lengths of all sections and the output are set to B_{accum} as defined by

$$B_{accum} = ceil(N_{sec\,s}(Log_2(D \times M)) + InputWordLength)$$

where N_{secs} is the number of filter sections.

Section fraction lengths and the fraction length of the output are set to the input fraction length.

Here is the display looks for this mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false
```

```
InputWordLength: 16
InputFracLength: 15

FilterInternals: 'FullPrecision'
```

Minimum Word Lengths

In minimum word length mode, you control the output word length explicitly. When the output word length is less than B_{accum}, roundoff noise is introduced at the output of the filter. Hogenauer's bit pruning theory (refer to [3] in the following References section) states that one valid design criterion is to make the word lengths of the different sections of the filter smaller than B_{accum} as well, so that the roundoff noise introduced by all sections does not exceed the roundoff noise introduced at the output.

In this mode, the design calculates the word lengths of each section to meet the Hogenauer criterion. The algorithm subtracts the number of bits computed using eq. 21 in Hogenauer's paper from B_{accum} to determine the word length each section.

To compute the fraction lengths of the different sections, the algorithm notes that the bits thrown out for this word length criterion are least significant bits (LSB), therefore each bit thrown out at a particular section decrements the fraction length of that section by one bit compared to the input fraction length. Setting the output word length for the filter automatically sets the output fraction length as well.

Here is the display for this mode:

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15
```

```
FilterInternals: 'MinWordLengths'
```

```
OutputWordLength: 16
```

Specify Word Lengths

In this mode, the design algorithm discards the LSBs, adjusting the fraction length so that unrecoverable overflow does not occur, always producing a reasonable output.

You can specify the word lengths for all sections and the output, but you cannot control the fraction lengths for those quantities.

To specify the word lengths, you enter a vector of length 2*(`NumberOfSections`), where each vector element represents the word length for a section. If you specify a calar, such as B_{accum}, the full-precision output word length, the algorithm expands that scalar to a vector of the appropriate size, applying the scalar value to each section.

The CIC design does not check that the specified word lengths are monotonically decreasing. There are some cases where the word lengths are not necessarily monotonically decreasing, for example

```
hcic=mfilt.cicdecim;
hcic.FilterInternals='minwordlengths';
hcic.Outputwordlength=14;
```

which are valid CIC filters but the word lengths do not decrease monotonically across the sections.

Here is the display looks like for the `SpecifyWordLengths` mode.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15
```

```
FilterInternals: 'SpecifyWordLengths'

SectionWordLengths: [19 18 18 17]

OutputWordLength: 16
```

Specify Precision

In this mode, you have full control over the word length and fraction lengths of all sections and the filter output.

When you elect the `SpecifyPrecision` mode, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the word length for each section. When you enter a scalar such as B_{accum}, the CIC algorithm expands that scalar to a vector of the appropriate size and applies the scalar value to each section and the output. The design does not check that this vector is monotonically decreasing.

Also, you must enter a vector of length 2*(`NumberOfSections`) with elements that represent the fraction length for each section as well. When you enter a calar such as B_{accum}, the design applies scalar expansion as done for the word lengths.

Here is the `SpecifyPrecision` display.

```
FilterStructure: 'Cascaded Integrator-Comb Decimator'
Arithmetic: 'fixed'
DifferentialDelay: 1
NumberOfSections: 2
DecimationFactor: 4
PersistentMemory: false

InputWordLength: 16
InputFracLength: 15

FilterInternals: 'SpecifyPrecision'

SectionWordLengths: [19 18 18 17]
SectionFracLengths: [14 13 13 12]
```

```
OutputWordLength: 16
OutputFracLength: 11
```

FilterStructure

Reports the type of filter object, such as a decimator or fractional integrator. You cannot set this property — it is always read only and results from your choice of `mfilt` object. Because of the length of the names of multirate filters, `FilterStructure` often returns a vector specification for the string. For example, when you use `mfilt.firfracinterp` to design a filter, `FilterStructure` returns as [1x49 char].

```
hm=mfilt.firfracinterp

hm =

         FilterStructure: [1x49 char]
               Numerator: [1x72 double]
        RateChangeFactors: [3 2]
         PersistentMemory: false
                   States: [24x1 double]
```

InputOffset

When you decimate signals whose length is not a multiple of the decimation factor M, the last samples — (nM +1) to [(n+1)(M) -1], where *n* is an integer — are processed and used to track where the filter stopped processing input data and when to expect the next output sample. If you think of the filtering process as generating an output for a block of input data, `InputOffset` contains a count of the number of samples in the last incomplete block of input data.

Note `InputOffset` applies only when you set `PersistentMemory` to `true`. Otherwise, `InputOffset` is not available for you to use.

Two different cases can arise when you decimate a signal:

1 The input signal is a multiple of the filter decimation factor. In this case, the filter processes the input samples and generates output samples for all inputs as determined by the decimation factor. For example, processing 99 input samples with a filter that decimates by three returns 33 output samples.

2 The input signal is not a multiple of the decimation factor. When this occurs, the filter processes all of the input samples, generates output samples as determined by the decimation factor, and has one or more input samples that were processed but did not generate an output sample.

For example, when you filter 100 input samples with a filter which has decimation factor of 3, you get 33 output samples, and 1 sample that did not generate an output. In this case, `InputOffset` stores the value 1 after the filter run.

`InputOffset` equal to 1 indicates that, if you divide your input signal into blocks of data with length equal to your filter decimation factor, the filter processed one sample from a new (incomplete) block of data. Subsequent inputs to the filter are concatenated with this single sample to form the next block of length m.

One way to define the value stored in `InputOffset` is

`InputOffset = mod(length(nx),m)`

where `nx` is the number of input samples in the data set and `m` is the decimation factor.

Storing `InputOffset` in the filter allows you to stop filtering a signal at any point and start over from there, provided that the `PersistentMemory` property is set to `true`. Being able to resume filtering after stopping a signal lets you break large data sets in to smaller pieces for filtering. With `PersistentMemory` set to `true` and the `InputOffset` property in the filter, breaking a signal into sections of arbitrary length and filtering the sections is equivalent to filtering the entire signal at once.

```
xtot=[x,x];
ytot=filter(hm1,xtot)
ytot =
```

```
          0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
reset(hm1);  % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hm1,x) filter(hm1,x)]


ysec =

          0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

InterpolationFactor

Amount to increase the sampling rate. Interpolation factor for the filter. It specifies the amount to increase the input sampling rate. It must be an integer. Two is the default value. You may use any positive value.

NumberOfSections

Number of sections used in the multirate filter. By default multirate filters use two sections, but any positive integer works.

OverflowMode

The `OverflowMode` property is specified as one of the following two strings indicating how to respond to overflows in fixed-point arithmetic:

- `'saturate'` — saturate overflows.

  When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the applicable word length and fraction length properties), these values are quantized to the value of either the largest or smallest representable value, depending on which is closest.

- `'wrap'` — wrap all overflows to the range of representable values.

  When the values of data to be quantized lie outside of the range of the largest and smallest representable numbers (as specified by the data format properties), these values are wrapped back into that range using modular

arithmetic relative to the smallest representable number. You can learn more about modular arithmetic in Fixed-Point Toolbox documentation.

These rules apply to the `OverflowMode` property.

- Applies to the accumulator and output data only.

- Does not apply to coefficients or input data. These always saturate the results.

- Does not apply to products. Products maintain full precision at all times. Your filters do not lose precision in the products.

Default value: `'saturate'`

Note Numbers in floating-point filters that extend beyond the dynamic range overflow to ±`inf`.

PolyphaseAccum

The idea behind `PolyphaseAccum` and `AccumWordLength/AccumFracLength` is to distinguish between the adders that always work in full precision (`PolyphaseAccum`) from the others [the adders that are controlled by the user (through `AccumWordLength` and `AccumFracLength`) and that may introduce quantization effects when you set property `FilterInternals` to `SpecifyPrecision`].

Given a product format determined by the input word and fraction lengths, and the coefficients word and fraction lengths, doing full precision accumulation means allowing enough guard bits to avoid overflows and underflows.

Property `PolyphaseAccum` stores the value that was in the accumulator the last time your filter ran out of input samples to process. The default value for `PolyphaseAccum` affects the next output only if `PersistentMemory` is `true` and `InputOffset` is not equal to 0.

`PolyphaseAccum` stores data in the format for the filter arithmetic. Double-precision filters store doubles in `PolyphaseAccum`. Single-precision

filter store singles in `PolyphaseAccum`. Fixed-point filters store `fi` objects in `PolyphaseAccum`.

PersistentMemory

Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter if you have not changed the filter since you constructed it. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected.

Determine whether the filter states get restored to their starting values for each filtering operation. The starting values are the values in place when you create the filter object. `PersistentMemory` returns to zero any state that the filter changes during processing. States that the filter does not change are not affected. Defaults to `true` — the filter retains memory about filtering operations from one to the next. Maintaining memory lets you filter large data sets as collections of smaller subsets and get the same result.

```
xtot=[x,x];
ytot=filter(hm1,xtot)
ytot =

        0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
reset(hm1);  % Clear history of the filter
hm1.PersistentMemory='true';
ysec=[filter(hm1,x) filter(hm1,x)]

ysec =

        0   -0.0003    0.0005   -0.0014    0.0028   -0.0054    0.0092
```

This test verifies that `ysec` (the signal filtered by sections) is equal to `ytot` (the entire signal filtered at once).

RateChangeFactors

Reports the decimation (m) and interpolation (l) factors for the filter object when you create fractional integrators and decimators, although m and l are used as arguments to both decimators and integrators, applying the

same meaning. Combining these factors as input arguments to the fractional decimator or integrator results in the final rate change for the signal.

For decimating filters, the default is [2,3]. For integrators, [3,2].

States

Stored conditions for the filter, including values for the integrator and comb sections. m is the differential delay and n is the number of sections in the filter.

About the States of Multirate Filters. In the states property you find the states for both the integrator and comb portions of the filter, stored in a filtstates object. states is a matrix of dimensions m+1-by-n, with the states in CIC filters apportioned as follows:

- States for the integrator portion of the filter are stored in the first row of the state matrix.

- States for the comb portion fill the remaining rows in the state matrix.

In the state matrix, state values are specified and stored in double format.

States stores conditions for the delays between each interpolator phase, the filter states, and the states at the output of each phase in the filter, including values for the interpolator and comb states.

The number of states is (lh-1)*m+(l-1)*(lo+mo) where lh is the length of each subfilter, and l and m are the interpolation and decimation factors. lo and mo, the input and output delays between each interpolation phase, are integers from Euclid's theorem such that lo*l-mo*m = -1 (refer to the reference for more details). Use euclidfactors to get lo and mo for an mfilt.firfracdecim object.

States defaults to a vector of zeros that has length equal to nstates(hm)

References for Multirate Filters

[1] Fliege, N.J., *Multirate Digital Signal Processing*, John Wiley and Sons, 1994.

[2] Harris, Fredric J, *Multirate Signal Processing for Communication Systems*, Prentice Hall PTR, 2004.

[3] Hogenauer, E. B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-29, No. 2, April 1981, pp. 155-162.

[4] Lyons, Richard G., *Understanding Digital Signal Processing*, Prentice Hall PTR, 2004

[5] Mitra, S.K., *Digital Signal Processing*, McGraw-Hill, 1998.

[6] Orfanidis, S.J., *Introduction to Signal Processing*, Prentice-Hall, Inc., 1996.

Glossary

This glossary defines terms related to fixed-point data types and numbers. These terms may appear in some or all of the documents that describe MathWorks® products that have fixed-point support.

arithmetic shift

Shift of the bits of a binary word for which the sign bit is recycled for each bit shift to the right. A zero is incorporated into the least significant bit of the word for each bit shift to the left. In the absence of overflows, each arithmetic shift to the right is equivalent to a division by 2, and each arithmetic shift to the left is equivalent to a multiplication by 2.

See also binary point, binary word, bit, logical shift, most significant bit

bias

Part of the numerical representation used to interpret a fixed-point number. Along with the slope, the bias forms the scaling of the number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

See also fixed-point representation, fractional slope, integer, scaling, slope, [Slope Bias]

binary number

Value represented in a system of numbers that has two as its base and that uses 1's and 0's (bits) for its notation.

See also bit

binary point

Symbol in the shape of a period that separates the integer and fractional parts of a binary number. Bits to the left of the binary point are integer bits and/or sign bits, and bits to the right of the binary point are fractional bits.

See also binary number, bit, fraction, integer, radix point

binary point-only scaling

Scaling of a binary number that results from shifting the binary point of the number right or left, and which therefore can only occur by powers of two.

See also binary number, binary point, scaling

binary word

Fixed-length sequence of bits (1's and 0's). In digital hardware, numbers are stored in binary words. The way in which hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

See also bit, data type, word

bit

Smallest unit of information in computer software or hardware. A bit can have the value 0 or 1.

ceiling (round toward)

Rounding mode that rounds to the closest representable number in the direction of positive infinity. This is equivalent to the `ceil` mode in Fixed-Point Toolbox software.

See also convergent rounding, floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

contiguous binary point

Binary point that occurs within the word length of a data type. For example, if a data type has four bits, its contiguous binary point must be understood to occur at one of the following five positions:

.0000

0.000

00.00

000.0

0000.

See also data type, noncontiguous binary point, word length

convergent rounding

Rounding mode that rounds to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 0.

See also ceiling (round toward), floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

data type

Set of characteristics that define a group of values. A fixed-point data type is defined by its word length, its fraction length, and whether it is signed or unsigned. A floating-point data type is defined by its word length and whether it is signed or unsigned.

See also fixed-point representation, floating-point representation, fraction length, signedness, word length

data type override

Parameter in the Fixed-Point Tool that allows you to set the output data type and scaling of fixed-point blocks on a system or subsystem level.

See also data type, scaling

exponent

Part of the numerical representation used to express a floating-point or fixed-point number.

1. Floating-point numbers are typically represented as

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

2. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

The exponent of a fixed-point number is equal to the negative of the fraction length:

$$exponent = -1 \times fraction\ length$$

See also bias, fixed-point representation, floating-point representation, fraction length, fractional slope, integer, mantissa, slope

fixed-point representation

Method for representing numerical values and data types that have a set range and precision.

1. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

The slope and the bias together represent the scaling of the fixed-point number.

2. Fixed-point data types can be defined by their word length, their fraction length, and whether they are signed or unsigned.

See also bias, data type, exponent, fraction length, fractional slope, integer, precision, range, scaling, slope, word length

floating-point representation

Method for representing numerical values and data types that can have changing range and precision.

1. Floating-point numbers can be represented as

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

2. Floating-point data types are defined by their word length.

See also data type, exponent, mantissa, precision, range, word length

floor (round toward)

Rounding mode that rounds to the closest representable number in the direction of negative infinity.

See also ceiling (round toward), convergent rounding, nearest (round toward), rounding, truncation, zero (round toward)

fraction

Part of a fixed-point number represented by the bits to the right of the binary point. The fraction represents numbers that are less than one.

See also binary point, bit, fixed-point representation

fraction length

Number of bits to the right of the binary point in a fixed-point representation of a number.

See also binary point, bit, fixed-point representation, fraction

fractional slope

Part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

The term *slope adjustment* is sometimes used as a synonym for fractional slope.

See also bias, exponent, fixed-point representation, integer, slope

full range

The broadest range available for a data type. From $-\infty$ to ∞ for floating-point types. For integer types, the representable range is the range from the smallest to largest integer value (finite) the type can represent. For example, from -128 to 127 for a signed 8–bit integer. Also known as representable range.

guard bits

Extra bits in either a hardware register or software simulation that are added to the high end of a binary word to ensure that no information is lost in case of overflow.

See also binary word, bit, overflow

incorrect range

A range that is too restrictive and does not include values that can actually occur in the model element. A range that is too broad is not considered incorrect because it will not lead to overflow.

See also range analysis

integer

1. Part of a fixed-point number represented by the bits to the left of the binary point. The integer represents numbers that are greater than or equal to one.

2. Also called the "stored integer." The raw binary number, in which the binary point is assumed to be at the far right of the word. The integer is part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

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

or

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

where the slope can be expressed as

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

See also bias, fixed-point representation, fractional slope, integer, real-world value, slope

integer length

Number of bits to the left of the binary point in a fixed-point representation of a number.

See also binary point, bit, fixed-point representation, fraction length, integer

least significant bit (LSB)

Bit in a binary word that can represent the smallest value. The LSB is the rightmost bit in a big-endian-ordered binary word. The weight of the LSB is related to the fraction length according to

$$weight\ of\ LSB = 2^{-fraction\ length}$$

See also big-endian, binary word, bit, most significant bit

logical shift

Shift of the bits of a binary word, for which a zero is incorporated into the most significant bit for each bit shift to the right and into the least significant bit for each bit shift to the left.

See also arithmetic shift, binary point, binary word, bit, most significant bit

mantissa

Part of the numerical representation used to express a floating-point number. Floating-point numbers are typically represented as

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

See also exponent, floating-point representation

model element

Entities in a model that range analysis software tracks, for example, blocks, signals, parameters, block internal data (such as accumulators, products).

See also range analysis

most significant bit (MSB)

Bit in a binary word that can represent the largest value. The MSB is the leftmost bit in a big-endian-ordered binary word.

See also binary word, bit, least significant bit

nearest (round toward)

Rounding mode that rounds to the closest representable number, with the exact midpoint rounded to the closest representable number in the direction of positive infinity. This is equivalent to the `nearest` mode in Fixed-Point Toolbox software.

See also ceiling (round toward), convergent rounding, floor (round toward), rounding, truncation, zero (round toward)

noncontiguous binary point

Binary point that is understood to fall outside the word length of a data type. For example, the binary point for the following 4-bit word is understood to occur two bits to the right of the word length,

0000__.

thereby giving the bits of the word the following potential values:

$2^5 2^4 2^3 2^2$ __.

See also binary point, data type, word length

one's complement representation

Representation of signed fixed-point numbers. Negating a binary number in one's complement requires a bitwise complement. That is, all 0's are flipped to 1's and all 1's are flipped to 0's. In one's complement notation there are two ways to represent zero. A binary word of all 0's represents "positive" zero, while a binary word of all 1's represents "negative" zero.

See also binary number, binary word, sign/magnitude representation, signed fixed-point, two's complement representation

overflow

Situation that occurs when the magnitude of a calculation result is too large for the range of the data type being used. In many cases you can choose to either saturate or wrap overflows.

See also saturation, wrapping

padding

Extending the least significant bit of a binary word with one or more zeros.

See also least significant bit

precision

1. Measure of the smallest numerical interval that a fixed-point data type and scaling can represent, determined by the value of the number's least significant bit. The precision is given by the slope, or the number

of fractional bits. The term *resolution* is sometimes used as a synonym for this definition.

2. Measure of the difference between a real-world numerical value and the value of its quantized representation. This is sometimes called quantization error or quantization noise.

See also data type, fraction, least significant bit, quantization, quantization error, range, slope

Q format

Representation used by Texas Instruments™ to encode signed two's complement fixed-point data types. This fixed-point notation takes the form

$Qm.n$

where

- *Q* indicates that the number is in Q format.

- *m* is the number of bits used to designate the two's complement integer part of the number.

- *n* is the number of bits used to designate the two's complement fractional part of the number, or the number of bits to the right of the binary point.

In Q format notation, the most significant bit is assumed to be the sign bit.

See also binary point, bit, data type, fixed-point representation, fraction, integer, two's complement

quantization

Representation of a value by a data type that has too few bits to represent it exactly.

See also bit, data type, quantization error

quantization error

Error introduced when a value is represented by a data type that has too few bits to represent it exactly, or when a value is converted from

one data type to a shorter data type. Quantization error is also called
quantization noise.

See also bit, data type, quantization

radix point

Symbol in the shape of a period that separates the integer and fractional
parts of a number in any base system. Bits to the left of the radix point
are integer and/or sign bits, and bits to the right of the radix point are
fraction bits.

See also binary point, bit, fraction, integer, sign bit

range

Span of numbers that a certain data type can represent.

See also data type, full range, precision, representable range

range analysis

Static analysis of model to derive minimum and maximum range values for elements in the model. The software statically analyzes the ranges of the individual computations in the model based on specified design ranges, inputs, and the semantics of the calculation.

real-world value

Stored integer value with fixed-point scaling applied. Fixed-point numbers can be represented as

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

or

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

where the slope can be expressed as

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

See also integer

representable range

The broadest range available for a data type. From $-\infty$ to ∞ for floating-point types. For integer types, the representable range is the range from the smallest to largest integer value (finite) the type can represent. For example, from -128 to 127 for a signed 8–bit integer. Also known as full range.

resolution

See **precision**

rounding

Limiting the number of bits required to express a number. One or more least significant bits are dropped, resulting in a loss of precision. Rounding is necessary when a value cannot be expressed exactly by the number of bits designated to represent it.

See also bit, ceiling (round toward), convergent rounding, floor (round toward), least significant bit, nearest (round toward), precision, truncation, zero (round toward)

saturation

Method of handling numeric overflow that represents positive overflows as the largest positive number in the range of the data type being used, and negative overflows as the largest negative number in the range.

See also overflow, wrapping

scaled double

A double data type that retains fixed-point scaling information. For example, in Simulink and Fixed-Point Toolbox software you can use data type override to convert your fixed-point data types to scaled doubles. You can then simulate to determine the ideal floating-point behavior of your system. After you gather that information you can turn data type override off to return to fixed-point data types, and your quantities still have their original scaling information because it was held in the scaled double data types.

scaling

1. Format used for a fixed-point number of a given word length and signedness. The slope and bias together form the scaling of a fixed-point number.

2. Changing the slope and/or bias of a fixed-point number without changing the stored integer.

See also bias, fixed-point representation, integer, slope

shift

Movement of the bits of a binary word either toward the most significant bit ("to the left") or toward the least significant bit ("to the right"). Shifts to the right can be either logical, where the spaces emptied at the front of the word with each shift are filled in with zeros, or arithmetic, where the word is sign extended as it is shifted to the right.

See also arithmetic shift, logical shift, sign extension

sign bit

Bit (or bits) in a signed binary number that indicates whether the number is positive or negative.

See also binary number, bit

sign extension

Addition of bits that have the value of the most significant bit to the high end of a two's complement number. Sign extension does not change the value of the binary number.

See also binary number, guard bits, most significant bit, two's complement representation, word

sign/magnitude representation

Representation of signed fixed-point or floating-point numbers. In sign/magnitude representation, one bit of a binary word is always the dedicated sign bit, while the remaining bits of the word encode the magnitude of the number. Negation using sign/magnitude representation consists of flipping the sign bit from 0 (positive) to 1 (negative), or from 1 to 0.

See also binary word, bit, fixed-point representation, floating-point representation, one's complement representation, sign bit, signed fixed-point, signedness, two's complement representation

signed fixed-point

Fixed-point number or data type that can represent both positive and negative numbers.

See also data type, fixed-point representation, signedness, unsigned fixed-point

signedness

The signedness of a number or data type can be signed or unsigned. Signed numbers and data types can represent both positive and negative values, whereas unsigned numbers and data types can only represent values that are greater than or equal to zero.

See also data type, sign bit, sign/magnitude representation, signed fixed-point, unsigned fixed-point

slope

Part of the numerical representation used to express a fixed-point number. Along with the bias, the slope forms the scaling of a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

See also bias, fixed-point representation, fractional slope, integer, scaling, [Slope Bias]

slope adjustment

See **fractional slope**

[Slope Bias]

Representation used to define the scaling of a fixed-point number.

See also bias, scaling, slope

stored integer

See **integer**

trivial scaling

Scaling that results in the real-world value of a number being simply equal to its stored integer value:

$$real\text{-}world\ value = stored\ integer$$

In [Slope Bias] representation, fixed-point numbers can be represented as

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

In the trivial case, slope = 1 and bias = 0.

In terms of binary point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$real\text{-}world\ value = stored\ integer \times 2^{-fraction\ length} = stored\ integer \times 2^0$$

Scaling is always trivial for pure integers, such as `int8`, and also for the true floating-point types `single` and `double`.

See also bias, binary point, binary point-only scaling, fixed-point representation, fraction length, integer, least significant bit, scaling, slope, [Slope Bias]

truncation

Rounding mode that drops one or more least significant bits from a number.

See also ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, zero (round toward)

two's complement representation

Common representation of signed fixed-point numbers. Negation using signed two's complement representation consists of a translation into one's complement followed by the binary addition of a one.

See also binary word, one's complement representation, sign/magnitude representation, signed fixed-point

unsigned fixed-point

Fixed-point number or data type that can only represent numbers greater than or equal to zero.

See also data type, fixed-point representation, signed fixed-point, signedness

word

Fixed-length sequence of binary digits (1's and 0's). In digital hardware, numbers are stored in words. The way hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

See also binary word, data type

word length

Number of bits in a binary word or data type.

See also binary word, bit, data type

wrapping

Method of handling overflow. Wrapping uses modulo arithmetic to cast a number that falls outside of the representable range the data type being used back into the representable range.

See also data type, overflow, range, saturation

zero (round toward)

Rounding mode that rounds to the closest representable number in the direction of zero. This is equivalent to the `fix` mode in Fixed-Point Toolbox software.

See also ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, truncation

Index