Communications System Toolbox™

Reference

R2013a

MATLAB®

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.

Communications System Toolbox™ Reference

Revision History

April 2011	First printing	Version 5.0
September 2011	Online only	Version 5.1 (R2011b)
March 2012	Online only	Version 5.2 (R2012a)
March 2012	Online only	Version 5.3 (R2012b)
March 2013	Online only	Version 5.4 (R2013a)

Contents

1 Functions — Alphabetical List

2 Blocks — Alphabetical List

3 Alphabetical List

Index

Functions — Alphabetical List

algdeintrlv

Purpose

Restore ordering of symbols using algebraically derived permutation table

Syntax

```
deintrlvd = algdeintrlv(data,num,'takeshita-costello',k,h)
deintrlvd = algdeintrlv(data,num,'welch-costas',alph)
```

Description

`deintrlvd = algdeintrlv(data,num,'takeshita-costello',k,h)` restores the original ordering of the elements in `data` using a permutation table that is algebraically derived using the Takeshita-Costello method. `num` is the number of elements in `data` if `data` is a vector, or the number of rows of `data` if `data` is a matrix with multiple columns. In the Takeshita-Costello method, `num` must be a power of 2. The multiplicative factor, `k`, must be an odd integer less than `num`, and the cyclic shift, `h`, must be a nonnegative integer less than `num`. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`deintrlvd = algdeintrlv(data,num,'welch-costas',alph)` uses the Welch-Costas method. In the Welch-Costas method, `num+1` must be a prime number. `alph` is an integer between 1 and `num` that represents a primitive element of the finite field GF(`num+1`).

To use this function as an inverse of the `algintrlv` function, use the same inputs in both functions, except for the `data` input. In that case, the two functions are inverses in the sense that applying `algintrlv` followed by `algdeintrlv` leaves data unchanged.

Examples

The code below uses the Takeshita-Costello method of `algintrlv` and `algdeintrlv`.

```
num = 16; % Power of 2
ncols = 3; % Number of columns of data to interleave
data = rand(num,ncols); % Random data to interleave
k = 3;
h = 4;
intdata = algintrlv(data,num,'takeshita-costello',k,h);
deintdata = algdeintrlv(intdata,num,'takeshita-costello',k,h);
```

References

[1] Heegard, Chris, and Stephen B. Wicker, *Turbo Coding*, Boston, Kluwer Academic Publishers, 1999.

[2] Takeshita, O. Y., and D. J. Costello, Jr., “New Classes Of Algebraic Interleavers for Turbo-Codes,” *Proc. 1998 IEEE International Symposium on Information Theory*, Boston, Aug. 16–21, 1998. p. 419.

See Also

`algintrlv`

How To

- “Interleaving”

algintrlv

Purpose

Reorder symbols using algebraically derived permutation table

Syntax

```
intrlvd = algintrlv(data,num,'takeshita-costello',k,h)
intrlvd = algintrlv(data,num,'welch-costas',alph)
```

Description

`intrlvd = algintrlv(data,num,'takeshita-costello',k,h)` rearranges the elements in `data` using a permutation table that is algebraically derived using the Takeshita-Costello method. `num` is the number of elements in `data` if `data` is a vector, or the number of rows of `data` if `data` is a matrix with multiple columns. In the Takeshita-Costello method, `num` must be a power of 2. The multiplicative factor, `k`, must be an odd integer less than `num`, and the cyclic shift, `h`, must be a nonnegative integer less than `num`. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`intrlvd = algintrlv(data,num,'welch-costas',alph)` uses the Welch-Costas method. In the Welch-Costas method, `num+1` must be a prime number. `alph` is an integer between 1 and `num` that represents a primitive element of the finite field GF(`num+1`). This means that every nonzero element of GF(`num+1`) can be expressed as `alph` raised to some integer power.

Examples

This example illustrates how to use the Welch-Costas method of algebraic interleaving.

1 Define `num` and the data to interleave.

```
num = 10; % Integer such that num+1 is prime
ncols = 3; % Number of columns of data to interleave
data = randi([0 num-1], num, ncols); % Random data to interleave
```

2 Find primitive polynomials of the finite field GF(`num+1`). The `gfprimfd` function represents each primitive polynomial as a row containing the coefficients in order of ascending powers.

```
pr = gfprimfd(1,'all',num+1) % Primitive polynomials of GF(num+1)
pr =
```

```
     3     1
     4     1
     5     1
     9     1
```

3 Notice from the output that `pr` has two columns and that the second column consists solely of 1s. In other words, each primitive polynomial is a monic degree-one polynomial. This is because `num+1` is prime. As a result, to find the primitive element that is a root of each primitive polynomial, find a root of the polynomial by subtracting the first column of `pr` from `num+1`.

```
primel = (num+1)-pr(:,1) % Primitive elements of GF(num+1)
primel =

     8
     7
     6
     2
```

4 Now define `alph` as one of the elements of `primel` and use `algintrlv`.

```
alph = primel(1); % Choose one primitive element.
intrlvd = algintrlv(data,num,'Welch-Costas',alph); % Interleave.
```

Algorithms

- A Takeshita-Costello interleaver uses a length-`num` cycle vector whose `n`th element is `mod(k*(n-1)*n/2, num)` for integers `n` between 1 and `num`. The function creates a permutation vector by listing, for each element of the cycle vector in ascending order, one plus the element's successor. The interleaver's actual permutation table is the result of shifting the elements of the permutation vector left by `h`. (The function performs all computations on numbers and indices modulo `num`.)
- A Welch-Costas interleaver uses a permutation that maps an integer `K` to `mod(A`^{K}`,num+1)-1`.

References

[1] Heegard, Chris, and Stephen B. Wicker, *Turbo Coding*, Boston, Kluwer Academic Publishers, 1999.

[2] Takeshita, O. Y., and D. J. Costello, Jr., “New Classes Of Algebraic Interleavers for Turbo-Codes,” *Proc. 1998 IEEE International Symposium on Information Theory*, Boston, Aug. 16–21, 1998. p. 419.

See Also

`algdeintrlv`

How To

- “Interleaving”

Purpose

Align two signals by delaying earliest signal

Syntax

```
[Xa,Ya] = alignsignals(X,Y)
[Xa,Ya] = alignsignals(X,Y,maxlag)
[Xa,Ya] = alignsignals(X,Y,maxlag,'truncate')
[Xa,Ya,D] = alignsignals( ___ )
```

Description

`[Xa,Ya] = alignsignals(X,Y)` estimates the delay *D* between the two input signals, `X` and`Y`, and returns the aligned signals, `Xa` and `Ya`.

- If `Y` is delayed with respect to `X`, then *D* is positive, and `X` is delayed by *D* samples.
- If `Y` is advanced with respect to `X`, then *D* is negative, and `Y` is delayed by –*D* samples.

Delays in `X` and `Y` can be introduced by prepending zeros.

`[Xa,Ya] = alignsignals(X,Y,maxlag)` uses `maxlag` as the maximum window size to find the estimated delay *D* between the two input signals, `X` and `Y`. It returns the aligned signals, `Xa` and `Ya`.

`[Xa,Ya] = alignsignals(X,Y,maxlag,'truncate')` keeps the lengths of the aligned signals, `Xa` and `Ya`, the same as those of the input signals, `X` and `Y`, respectively.

- If the estimated delay *D* is positive, then *D* zeros are prepended to `X` and the last *D* samples of `X` are truncated.
- If the estimated delay *D* is negative, then –*D* zeros are prepended to `Y` and the last –*D* samples of `Y` are truncated.

Notes `X` and `Y` are row or column vectors of length *LX* and *LY*, respectively.

- If $D \geq LX$, then `Xa` consists of *LX* zeros. All samples of `X` are lost.
- If $-D \geq LY$, then `Ya` consists of *LY* zeros. All samples of `Y` are lost.

To avoid assigning a specific value to `maxlag` when using the `'truncate'` option, set `maxlag` to `[]`.

`[Xa,Ya,D] = alignsignals(___)` returns the estimated delay `D`. This syntax can include any of the input arguments used in previous syntaxes.

Input Arguments

X - First input signal

vector of numeric values

First input signal, specified as a numeric vector of length *LX*.

Example: `[1,2,3]`

Data Types
`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`
Complex Number Support: Yes

Y - Second input signal

vector of numeric values

Second input signal, specified as a numeric vector of length *LY*.

Example: `[0,0,1,2,3]`

Data Types
`single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`
Complex Number Support: Yes

maxlag - Maximum window size or lag

scalar integer | []

Maximum window size, or lag, specified as an integer-valued scalar. By default, `maxlag` is equal to `max(length(X),length(Y))-1`. If `maxlag` is input as `[]`, it is replaced by the default value. If `maxlag` is negative, it is replaced by its absolute value. If `maxlag` is not integer valued, or is complex, `Inf`, or `NaN`, then `alignsignals` returns an error.

Example: `2`

Data Types

`single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Default: `max(length(X),length(Y))-1`

Output Arguments

Xa - Aligned first signal

vector of numeric values

Aligned first signal, returned as a numeric vector that is aligned with the second output argument `Ya`. If input argument `X` is a row vector, then `Xa` will also be a row vector. If input argument `X` is a column vector, then `Xa` will also be a column vector. If you specify the `'truncate'` option and the estimated delay D is positive, then `Xa` is equivalent to the input signal `X` with D zeros prepended to it and its last D samples truncated.

Ya - Aligned second signal

vector of numeric values

Aligned second signal, returned as a numeric vector that is aligned with the first output argument `Xa`. If input argument `Y` is a row vector, then `Ya` is also a row vector. If input argument `Y` is a column vector, then `Ya` is also a column vector. If you specify the `'truncate'` option and the estimated delay D is negative, then `Ya` is equivalent to the input signal `Y` with $-D$ zeros prepended to it and its last $-D$ samples truncated.

D - Estimated delay between input signals

scalar integer

Estimated delay between input signals, returned as a scalar integer. This integer represents the number of samples by which the two input signals, X and Y are offset.

- If Y is delayed with respect to X, then D is positive and X is delayed by D samples.
- If Y is advanced with respect to X, then D is negative and Y is delayed by –D samples.
- If X and Y are already aligned, then D is zero and neither X nor Y are delayed.

If you specify a value for the input argument maxlag, then D must be less than or equal to maxlag.

Examples

Aligning two signals where the second signal lags by two samples

Align signal X when Y is delayed with respect to X by two samples.

Create two signals, X and Y. Y is exactly the same as X, except Y has two leading zeros. Align the two signals.

```
X = [1 2 3];
Y = [0 0 1 2 3];
MAXLAG = 2;
[Xa Ya D] = alignsignals(X, Y, MAXLAG)
```

The resulting values are:

```
Xa =
     0     0     1     2     3
Ya =
     0     0     1     2     3
D =
     2
```

Aligning two signals where the first signal lags by three samples

Align signal Y with respect to X by advancing it three samples.

Create two signals, X and Y. X is exactly the same as Y, except X has three leading zeros and one additional following zero. Align the two signals.

```
X = [0 0 0 1 2 3 0 0]';
Y = [1 2 3 0]';
[Xa Ya] = alignsignals(X, Y)
```

The resulting values are:

```
Xa =
     0
     0
     0
     1
     2
     3
     0
     0
Ya =
     0
     0
     0
     1
     2
     3
     0
```

Aligning two signals where the second signal is noisy

Align signal Y with respect to X, despite the fact that Y is a noisy signal.

Create two signals, X and Y. Y is exactly the same as X with some noise added to it. Align the two signals.

```
X = [0 0 1 2 3 0];
```

```
Y = [0.02 0.12 1.08 2.21 2.95 -0.09];
[Xa Ya D] = alignsignals(X, Y)
```

You do not need to change the input signals to produce the output signals. The delay D is zero. The resulting values are:

```
Xa =
     0     0     1     2     3     0
Ya =
    0.0200    0.1200    1.0800    2.2100    2.9500   -0.0900
D =
     0
```

Aligning two signals where the second signal is a periodic repetition of the first signal

Align signal Y with respect to X, despite the fact that Y is a periodic repetition of X. Return the smallest possible delay.

Create two signals, X and Y. Y is exactly the same as X with some noise added to it. Align the two signals.

```
X = [0 1 2 3];
Y = [1 2 3 0 0 0 0 1 2 3 0 0];
[Xa Ya D] = alignsignals(X, Y)
```

The resulting values are:

```
Xa =
     0     1     2     3
Ya =
     0     1     2     3     0     0     0     0     1     2     3     0
D =
    -1
```

Aligning two signals using the 'truncate' option

Invoke the 'truncate' option when calling the alignsignals function.

Create two signals, `X` and `Y`. `Y` is exactly the same as `X`, except `Y` has two leading zeros. Align the two signals, applying the `'truncate'` directive.

```
X = [1 2 3];
Y = [0 0 1 2 3];
[Xa Ya D] = alignsignals(X, Y, [], 'truncate');
```

Observe that the output signal `Xa` has a length of 3, the same length as input signal `X`. The resulting values are:

```
Xa =
     0     0     1
Ya =
     0     0     1     2     3
D =
     2
```

In the case where using the `'truncate'` option ends up truncating all the original data of `X`, a warning is issued. To make `alignsignals` issue such a warning, run the following example.

```
X = [1 2 3];
Y = [0 0 0 0 1 2 3];
[Xa Ya D] = alignsignals(X, Y, [], 'truncate')
```

The resulting warning is:

```
Warning: All original data in the first input X has been
truncated because the length of X is smaller than the
estimated delay D: to avoid truncating this data do not use
the 'trunc' option.
> In alignsignals at 136
```

The resulting values are:

```
Xa =
     0     0     0
Ya =
     0     0     0     0     1     2     3
```

```
D =
     4
```

Algorithms

- You can find the theory on delay estimation in the specification of the `finddelay` function (see "Algorithms" on page 1-264).
- The `alignsignals` function uses the estimated delay *D* to delay the earliest signal such that the two signals have the same starting point.
- As specified for the `finddelay` function, the pair of signals need not be exact delayed copies of each other. However, the signals can be successfully aligned only if there is sufficient correlation between them.

For more information on estimating covariance and correlation functions, see [1].

References

[1] Orfanidis, S.J., *Optimum Signal Processing. An Introduction. 2nd Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

See Also

`finddelay`

Related Examples

- "Use the Find Delay and Align Signals Blocks"

Concepts

- "Delays"

Purpose

Amplitude demodulation

Syntax

```
z = amdemod(y,Fc,Fs)
z = amdemod(y,Fc,Fs,ini_phase)
z = amdemod(y,Fc,Fs,ini_phase,carramp)
z = amdemod(y,Fc,Fs,ini_phase,carramp,num,den)
```

Description

`z = amdemod(y,Fc,Fs)` demodulates the amplitude modulated signal `y` from a carrier signal with frequency `Fc` (Hz). The carrier signal and `y` have sample frequency `Fs` (Hz). The modulated signal `y` has zero initial phase and zero carrier amplitude, so it represents suppressed carrier modulation. The demodulation process uses the lowpass filter specified by `[num,den] = butter(5,Fc*2/Fs)`.

Note The `Fc` and `Fs` arguments must satisfy `Fs` > 2(`Fc` + `BW`), where `BW` is the bandwidth of the original signal that was modulated.

`z = amdemod(y,Fc,Fs,ini_phase)` specifies the initial phase of the modulated signal in radians.

`z = amdemod(y,Fc,Fs,ini_phase,carramp)` demodulates a signal that was created via transmitted carrier modulation instead of suppressed carrier modulation. `carramp` is the carrier amplitude of the modulated signal.

`z = amdemod(y,Fc,Fs,ini_phase,carramp,num,den)` specifies the numerator and denominator of the lowpass filter used in the demodulation.

Examples

The code below illustrates the use of a nondefault filter.

```
t = .01;
Fc = 10000; Fs = 80000;
t = [0:1/Fs:0.01]';
s = sin(2*pi*300*t)+2*sin(2*pi*600*t); % Original signal
```

```
[num,den] = butter(10,Fc*2/Fs); % Lowpass filter

y1 = ammod(s,Fc,Fs); % Modulate.
s1 = amdemod(y1,Fc,Fs,0,0,num,den); % Demodulate.
```

See Also

ammod | ssbdemod | fmdemod | pmdemod

How To

- “Digital Modulation”

Purpose

Amplitude modulation

Syntax

```
y = ammod(x,Fc,Fs)
y = ammod(x,Fc,Fs,ini_phase)
y = ammod(x,Fc,Fs,ini_phase,carramp)
```

Description

`y = ammod(x,Fc,Fs)` uses the message signal `x` to modulate a carrier signal with frequency `Fc` (Hz) using amplitude modulation. The carrier signal and `x` have sample frequency `Fs` (Hz). The modulated signal has zero initial phase and zero carrier amplitude, so the result is suppressed-carrier modulation.

Note The `x`, `Fc`, and `Fs` input arguments must satisfy `Fs` > 2(`Fc` + `BW`), where `BW` is the bandwidth of the modulating signal `x`.

`y = ammod(x,Fc,Fs,ini_phase)` specifies the initial phase in the modulated signal `y` in radians.

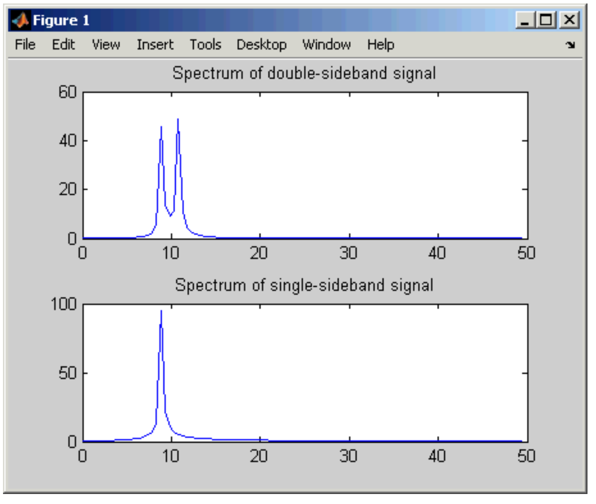
`y = ammod(x,Fc,Fs,ini_phase,carramp)` performs transmitted-carrier modulation instead of suppressed-carrier modulation. The carrier amplitude is `carramp`.

Examples

The example below compares double-sideband and single-sideband amplitude modulation.

```
% Sample the signal 100 times per second, for 2 seconds.
Fs = 100;
t = [0:2*Fs+1]'/Fs;
Fc = 10; % Carrier frequency
x = sin(2*pi*t); % Sinusoidal signal

% Modulate x using single- and double-sideband AM.
ydouble = ammod(x,Fc,Fs);
ysingle = ssbmod(x,Fc,Fs);
```

```
% Compute spectra of both modulated signals.
zdouble = fft(ydouble);
zdouble = abs(zdouble(1:length(zdouble)/2+1));
frqdouble = [0:length(zdouble)-1]*Fs/length(zdouble)/2;
zsingle = fft(ysingle);
zsingle = abs(zsingle(1:length(zsingle)/2+1));
frqsingle = [0:length(zsingle)-1]*Fs/length(zsingle)/2;

% Plot spectra of both modulated signals.
figure;
subplot(2,1,1); plot(frqdouble,zdouble);
title('Spectrum of double-sideband signal');
subplot(2,1,2); plot(frqsingle,zsingle);
title('Spectrum of single-sideband signal');
```



See Also

amdemod | ssbmod | fmmod | pmmod

How To

- “Digital Modulation”

Purpose

Decode binary code using arithmetic decoding

Syntax

```
dseq = arithdeco(code,counts,len)
```

Description

`dseq = arithdeco(code,counts,len)` decodes the binary arithmetic code in the vector `code` to recover the corresponding sequence of `len` symbols. The vector `counts` represents the source's statistics by listing the number of times each symbol of the source's alphabet occurs in a test data set. This function assumes that the data in `code` was produced by the `arithenco` function.

Examples

This example is similar to the example on the `arithenco` reference page, except that it uses `arithdeco` to recover the original sequence.

```
counts = [99 1]; % A one occurs 99% of the time.
len = 1000;
seq = randsrc(1,len,[1 2; .99 .01]); % Random sequence
code = arithenco(seq,counts);
dseq = arithdeco(code,counts,length(seq)); % Decode.
isequal(seq,dseq) % Check that dseq matches the original seq.
```

The output is

```
ans =

     1
```

Algorithms

This function uses the algorithm described in [1].

References

[1] Sayood, Khalid, *Introduction to Data Compression,* San Francisco, Morgan Kaufmann, 2000.

See Also

`arithenco`

How To

- "Arithmetic Coding"

Purpose

Encode sequence of symbols using arithmetic coding

Syntax

```
code = arithenco(seq,counts)
```

Description

`code = arithenco(seq,counts)` generates the binary arithmetic code corresponding to the sequence of symbols specified in the vector `seq`. The vector `counts` represents the source's statistics by listing the number of times each symbol of the source's alphabet occurs in a test data set.

Examples

This example illustrates the compression that arithmetic coding can accomplish in some situations. A source has a two-symbol alphabet and produces a test data set in which 99% of the symbols are 1s. Encoding 1000 symbols from this source produces a code vector having many fewer than 1000 elements. The actual number of elements in `code` varies, depending on the particular random sequence contained in `seq`.

```
counts = [99 1]; % A one occurs 99% of the time.
len = 1000;
seq = randsrc(1,len,[1 2; .99 .01]); % Random sequence
code = arithenco(seq,counts);
s = size(code) % length of code is only 8.3% of length of seq.
```

The output is

```
s =

     1    83
```

Algorithms

This function uses the algorithm described in [1].

References

[1] Sayood, Khalid, *Introduction to Data Compression,* San Francisco, Morgan Kaufmann, 2000.

See Also

`arithdeco`

How To

- “Arithmetic Coding”

Purpose Add white Gaussian noise to signal

Syntax

```
y = awgn(x,snr)
y = awgn(x,snr,sigpower)
y = awgn(x,snr,'measured')
y = awgn(x,snr,sigpower,s)
y = awgn(x,snr,'measured',state)
y = awgn(...,powertype)
```

Description

`y = awgn(x,snr)` adds white Gaussian noise to the vector signal `x`. The scalar `snr` specifies the signal-to-noise ratio per sample, in dB. If `x` is complex, `awgn` adds complex noise. This syntax assumes that the power of `x` is 0 dBW.

`y = awgn(x,snr,sigpower)` is the same as the syntax above, except that `sigpower` is the power of `x` in dBW.

`y = awgn(x,snr,'measured')` is the same as `y = awgn(x,snr)`, except that `awgn` measures the power of `x` before adding noise.

`y = awgn(x,snr,sigpower,s)` uses s, which is a random stream handle, to generate random noise samples with randn. If s is an integer, then resets the state of randn to s. The latter usage is obsolete and may be removed in a future release. If you want to generate repeateable noise samples, then provide the handle of a random stream or use reset method on the default random stream.

`y = awgn(x,snr,'measured',state)` is the same as `y = awgn(x,snr,'measured')`, except that `awgn` first resets the state of normal random number generator `randn` to the integer `state`.

Note This usage is deprecated and may be removed in a future release. Instead of `state`, use `s`, as in the previous example.

`y = awgn(...,`*`powertype`*`)` is the same as the previous syntaxes, except that the string *`powertype`* specifies the units of `snr` and `sigpower`. Choices for *`powertype`* are `'db'` and `'linear'`. If *`powertype`* is `'db'`, then `snr` is measured in dB and `sigpower` is measured in dBW. If *powertype* is `'linear'`, `snr` is measured as a ratio and `sigpower` is measured in watts.

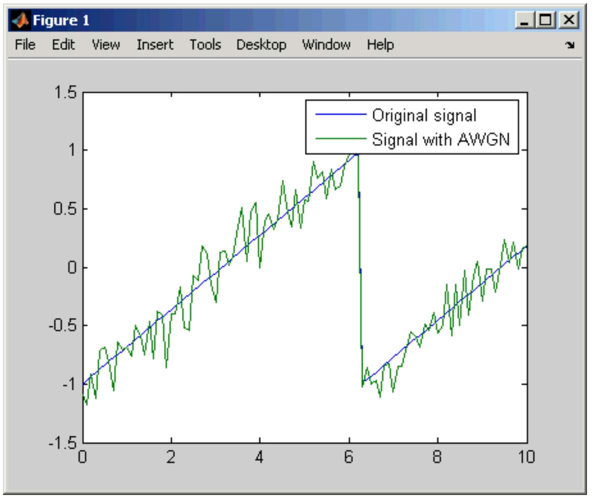
Relationship Among SNR, E_s/N_0, and E_b/N_0

For the relationships between SNR and other measures of the relative power of the noise, see “AWGN Channel Noise Level”.

Examples

The commands below add white Gaussian noise to a sawtooth signal. It then plots the original and noisy signals.

```
t = 0:.1:10;
x = sawtooth(t); % Create sawtooth signal.
y = awgn(x,10,'measured'); % Add white Gaussian noise.
plot(t,x,t,y) % Plot both signals.
legend('Original signal','Signal with AWGN');
```



The `scattereyedemo` also illustrates the use of the `awgn` function.

See Also

`wgn` | `randn` | `bsc`

bchdec

Purpose

BCH decoder

Syntax

```
decoded = bchdec(code,n,k)
decoded = bchdec(...,paritypos)
[decoded,cnumerr] = bchdec(...)
[decoded,cnumerr,ccode] = bchdec(...)
```

Description

`decoded = bchdec(code,n,k)` attempts to decode the received signal in `code` using an [n,k] BCH decoder with the narrow-sense generator polynomial. `code` is a Galois array of symbols over GF(2). Each `n`-element row of `code` represents a corrupted systematic codeword, where the parity symbols are at the end and the leftmost symbol is the most significant symbol.

In the Galois array `decoded`, each row represents the attempt at decoding the corresponding row in `code`. A *decoding failure* occurs if `bchdec` detects more than t errors in a row of `code`, where t is the number of correctable errors as reported by `bchgenpoly`. In the case of a decoding failure, `bchdec` forms the corresponding row of `decoded` by merely removing `n-k` symbols from the end of the row of `code`.

`decoded = bchdec(...,`*`paritypos`*`)` specifies whether the parity symbols in `code` were appended or prepended to the message in the coding operation. The string *`paritypos`* can be either `'end'` or `'beginning'`. The default is `'end'`. If *`paritypos`* is `'beginning'`, then a decoding failure causes `bchdec` to remove `n-k` symbols from the beginning rather than the end of the row.

`[decoded,cnumerr] = bchdec(...)` returns a column vector `cnumerr`, each element of which is the number of corrected errors in the corresponding row of `code`. A value of `-1` in `cnumerr` indicates a decoding failure in that row in `code`.

`[decoded,cnumerr,ccode] = bchdec(...)` returns `ccode`, the corrected version of `code`. The Galois array `ccode` has the same format as `code`. If a decoding failure occurs in a certain row of `code`, the corresponding row in `ccode` contains that row unchanged.

Results of Error Correction

BCH decoders correct up to a certain number of errors, specified by the user. If the input contains more errors than the decoder is meant to correct, the decoder will most likely not output the correct codeword.

The chance of a BCH decoder decoding a corrupted input to the correct codeword depends on the number of errors in the input and the number of errors the decoder is meant to correct.

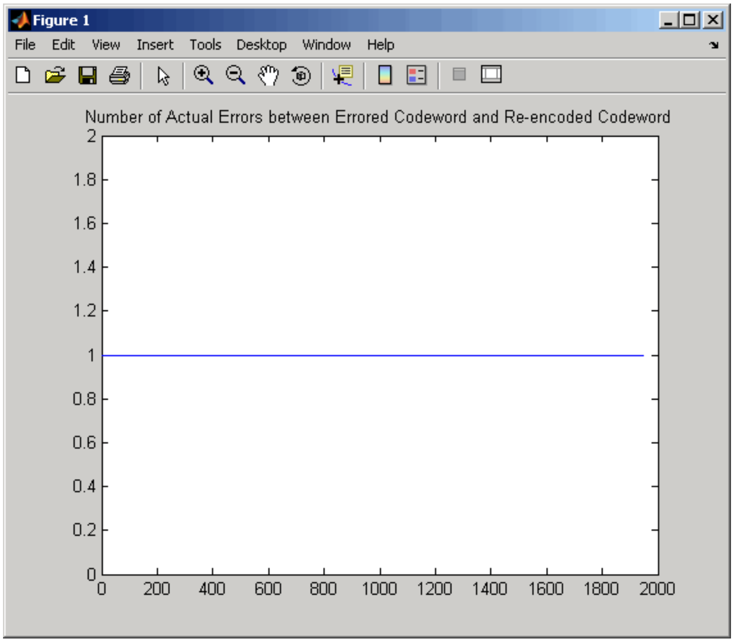
For example, when a single-error-correcting BCH decoder is given input with two errors, it actually decodes it to a different codeword. When a double-error-correcting BCH decoder is given input with three errors, then it only sometimes decodes it to a valid codeword.

The following code illustrates this phenomenon for a single-error-correcting BCH decoder given input with two errors.

```
n = 63; k = 57;
s = RandStream('swb2712', 'Seed', 9973);
msg = gf(randi(s,[0 1],1,k));
code = bchenc(msg, n, k);

% Add 2 errors
cnumerr2 = zeros(nchoosek(n,2),1);
nErrs = zeros(nchoosek(n,2),1);
cnumerrIdx = 1;
for idx1 = 1 : n-1
    sprintf('idx1 for 2 errors = %d', idx1)
    for idx2 = idx1+1 : n
        errors = zeros(1,n);
        errors(idx1) = 1;
        errors(idx2) = 1;
        erroredCode = code + gf(errors);
        [decoded2, cnumerr2(cnumerrIdx)]...
          = bchdec(erroredCode, n, k);

        % If bchdec thinks it corrected only one error,
        % then encode the decoded message.  Check that
        % the re-encoded message differs from the errored
        % message in only one coordinate.
```

```
        if cnumerr2(cnumerrIdx) == 1
            code2 = bchenc(decoded2, n, k);
            nErrs(cnumerrIdx) = biterr(double(erroredCode.x),...
              double(code2.x));
        end

        cnumerrIdx = cnumerrIdx + 1;
    end
end

% Plot the computed number of errors, based on the difference
% between the double-errored codeword and the codeword that was
% re-encoded from the initial decoding.
plot(nErrs)
title(['Number of Actual Errors between Errored Codeword and' ...
 'Re-encoded Codeword'])
```

The resulting plot shows that all inputs with two errors are decoded to a codeword that differs in exactly one position.



Examples

The script below encodes a (random) message, simulates the addition of noise to the code, and then decodes the message.

```
m = 4; n = 2^m-1; % Codeword length
k = 5; % Message length
nwords = 10; % Number of words to encode
msg = gf(randi([0 1],nwords,k));
% Find t, the error-correction capability.
[genpoly,t] = bchgenpoly(n,k);
% Define t2, the number of errors to add in this example.
t2 = t;

% Encode the message.
code = bchenc(msg,n,k);
% Corrupt up to t2 bits in each codeword.
noisycode = code + randerr(nwords,n,1:t2);
```

```
% Decode the noisy code.
[newmsg,err,ccode] = bchdec(noisycode,n,k);
if ccode==code
   disp('All errors were corrected.')
end
if newmsg==msg
   disp('The message was recovered perfectly.')
end
```

In this case, all errors are corrected and the message is recovered perfectly. However, if you change the definition of `t2` to

```
t2 = t+1;
```

then some codewords will contain more than `t` errors. This is too many errors, and some are not corrected.

Algorithms

`bchdec` uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the works listed in "References" on page 1-28.

Limitations

The maximum allowable value of `n` is 65535.

References

[1] Wicker, Stephen B., *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, NJ, Prentice Hall, 1995.

[2] Berlekamp, Elwyn R., *Algebraic Coding Theory*, New York, McGraw-Hill, 1968.

See Also

`bchenc` | `bchgenpoly`

How To

- "Block Codes"

Purpose

BCH encoder

Syntax

```
code = bchenc(msg,n,k)
code = bchenc(...,paritypos)
```

Description

`code = bchenc(msg,n,k)` encodes the message in `msg` using an [`n`,`k`] BCH encoder with the narrow-sense generator polynomial. `msg` is a Galois array of symbols over GF(2). Each `k`-element row of `msg` represents a message word, where the leftmost symbol is the most significant symbol. Parity symbols are at the end of each word in the output Galois array `code`.

`code = bchenc(...,`*`paritypos`*`)` specifies whether `bchenc` appends or prepends the parity symbols to the input message to form `code`. The string *`paritypos`* can be either `'end'` or `'beginning'`. The default is `'end'`.

The tables below list valid [`n`,`k`] pairs for small values of `n`, as well as the corresponding values of the error-correction capability, `t`.

n	k	t
7	4	1

n	k	t
15	11	1
15	7	2
15	5	3

n	k	t
31	26	1
31	21	2
31	16	3
31	11	5
31	6	7

n	k	t
63	57	1
63	51	2
63	45	3
63	39	4
63	36	5
63	30	6
63	24	7
63	18	10
63	16	11
63	10	13
63	7	15

n	k	t
127	120	1
127	113	2

n	k	t
127	106	3
127	99	4
127	92	5
127	85	6
127	78	7
127	71	9
127	64	10
127	57	11
127	50	13
127	43	14
127	36	15
127	29	21
127	22	23
127	15	27
127	8	31

n	k	t
255	247	1
255	239	2
255	231	3
255	223	4
255	215	5
255	207	6

n	k	t
255	199	7
255	191	8
255	187	9
255	179	10
255	171	11
255	163	12
255	155	13
255	147	14
255	139	15
255	131	18
255	123	19
255	115	21
255	107	22
255	99	23
255	91	25
255	87	26
255	79	27
255	71	29
255	63	30
255	55	31
255	47	42
255	45	43
255	37	45
255	29	47
255	21	55

n	k	t
255	13	59
255	9	63

n	k	t
511	502	1
511	493	2
511	484	3
511	475	4
511	466	5
511	457	6
511	448	7
511	439	8
511	430	9
511	421	10
511	412	11
511	403	12
511	394	13
511	385	14
511	376	15
511	367	16
511	358	18
511	349	19
511	340	20

n	k	t
511	331	21
511	322	22
511	313	23
511	304	25
511	295	26
511	286	27
511	277	28
511	268	29
511	259	30
511	250	31
511	241	36
511	238	37
511	229	38
511	220	39
511	211	41
511	202	42
511	193	43
511	184	45
511	175	46
511	166	47
511	157	51
511	148	53
511	139	54
511	130	55
511	121	58

n	k	t
511	112	59
511	103	61
511	94	62
511	85	63
511	76	85
511	67	87
511	58	91
511	49	93
511	40	95
511	31	109
511	28	111
511	19	119
511	10	121

Examples

See the example on the reference page for the function `bchdec`.

Limitations

The maximum allowable value of `n` is 65535.

See Also

`bchdec` | `bchgenpoly` | `bchnumerr`

How To

- "Block Codes"

bchgenpoly

Purpose

Generator polynomial of BCH code

Syntax

```
genpoly = bchgenpoly(n,k)
genpoly = bchgenpoly(n,k,prim_poly)
genpoly = bchgenpoly(n,k,prim_poly,outputFormat)
[genpoly,t] = bchgenpoly(...)
```

Description

`genpoly = bchgenpoly(n,k)` returns the narrow-sense generator polynomial of a BCH code with codeword length `n` and message length `k`. The codeword length `n` must have the form 2^m-1 for some integer m between 3 and 16. The output `genpoly` is a Galois row vector that represents the coefficients of the generator polynomial in order of descending powers. The narrow-sense generator polynomial is LCM[m_1(x), m_2(x), ..., m_2t(x)], where:

- LCM represents the least common multiple,
- m_i(x) represents the minimum polynomial corresponding to α^i, α is a root of the default primitive polynomial for the field GF(`n+1`),
- and t represents the error-correcting capability of the code.

Note Although the `bchgenpoly` function performs intermediate computations in GF(`n+1`), the final polynomial has binary coefficients. The output from `bchgenpoly` is a Galois vector in GF(2) rather than in GF(`n+1`).

`genpoly = bchgenpoly(n,k,prim_poly)` is the same as the syntax above, except that `prim_poly` specifies the primitive polynomial for GF(`n+1`) that has Alpha as a root. `prim_poly` is an integer whose binary representation indicates the coefficients of the primitive polynomial in order of descending powers. To use the default primitive polynomial for GF(`n+1`), set `prim_poly` to `[]`.

`genpoly = bchgenpoly(n,k,prim_poly,outputFormat)` is the same as the previous syntax, except that `outputFormat` specifies output data type. The value of `outputFormat` can be ‘`gf`’ or ’`double`’ corresponding to Galois field and double data types respectively. The default value of `outputFormat` is 'gf'.

`[genpoly,t] = bchgenpoly(...)` returns `t`, the error-correction capability of the code.

Examples

The results below show that a [15,11] BCH code can correct one error and has a generator polynomial $X^4 + X + 1$.

```
m = 4;
n = 2^m-1; % Codeword length
k = 11; % Message length
% Get generator polynomial and error-correction capability.
[genpoly,t] = bchgenpoly(n,k)
```

The output is

```
genpoly = GF(2) array.

Array elements =

     1     0     0     1     1


t =

     1
```

Limitations

The maximum allowable value of `n` is 511.

References

[1] Peterson, W. Wesley, and E. J. Weldon, Jr., *Error-Correcting Codes*, 2nd ed., Cambridge, MA, MIT Press, 1972.

See Also

`bchenc` | `bchdec` | `bchnumerr`

How To

- "Block Codes"

bchnumerr

Purpose

Number of correctable errors for BCH code

Syntax

```
T = bchnumerr(N)
T = bchnumerr(N, K)
```

Description

`T = bchnumerr(N)` returns all the possible combinations of message length, `K`, and number of correctable errors, `t`, for a BCH code of codeword length, `N`. `N` must have the form 2^m-1 for some integer, m, between 3 and 16. `T` is a matrix with three columns. The first column lists `N`, the second column lists `K`, and the third column lists `t`.

`T = bchnumerr(N, K)` returns the number of correctable errors, `t`, for an `(N, K)` BCH code.

See Also

`bchenc` | `bchdec` | `bchgenpoly`

Purpose

Bit error rate (BER) for uncoded AWGN channels

Syntax

```
ber = berawgn(EbNo,'pam',M)
ber = berawgn(EbNo,'qam',M)
ber = berawgn(EbNo,'psk',M,dataenc)
ber = berawgn(EbNo,'oqpsk',dataenc)
ber = berawgn(EbNo,'dpsk',M)
ber = berawgn(EbNo,'fsk',M,coherence)
ber = berawgn(EbNo,'fsk',2,coherence,rho)
ber = berawgn(EbNo,'msk',precoding)
ber = berawgn(EbNo,'msk',precoding,coherence)
berlb = berawgn(EbNo,'cpfsk',M,modindex,kmin)
[BER,SER] = berawgn(EbNo, ...)
```

Alternatives

As an alternative to the `berawgn` function, invoke the BERTool GUI (`bertool`), and use the **Theoretical** tab.

Description

For All Syntaxes

The `berawgn` function returns the BER of various modulation schemes over an additive white Gaussian noise (AWGN) channel. The first input argument, `EbNo`, is the ratio of bit energy to noise power spectral density, in dB. If `EbNo` is a vector, the output `ber` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels. The supported modulation schemes, which correspond to the second input argument to the function, are in the following table.

Modulation Scheme	Second Input Argument
Phase shift keying (PSK)	`'psk'`
Offset quaternary phase shift keying (OQPSK)	`'oqpsk'`
Differential phase shift keying (DPSK)	`'dpsk'`
Pulse amplitude modulation (PAM)	`'pam'`

Modulation Scheme	Second Input Argument
Quadrature amplitude modulation (QAM)	`'qam'`
Frequency shift keying (FSK)	`'fsk'`
Minimum shift keying (MSK)	`'msk'`
Continuous phase frequency shift keying (CPFSK)	`'cpfsk'`

Most syntaxes also have an `M` input that specifies the alphabet size for the modulation. `M` must have the form 2^k for some positive integer `k`. For all cases, the function assumes the use of a Gray-coded signal constellation.

For Specific Syntaxes

`ber = berawgn(EbNo,'pam',M)` returns the BER of uncoded PAM over an AWGN channel with coherent demodulation.

`ber = berawgn(EbNo,'qam',M)` returns the BER of uncoded QAM over an AWGN channel with coherent demodulation. The alphabet size, `M`, must be at least 4. When $k = \log_2 M$ is odd, a rectangular constellation of size $M = I \times J$ is used, where $I = 2^{\frac{k-1}{2}}$ and $J = 2^{\frac{k+1}{2}}$.

`ber = berawgn(EbNo,'psk',M,`*`dataenc`*`)` returns the BER of coherently detected uncoded PSK over an AWGN channel. *`dataenc`* is either `'diff'` for differential data encoding or `'nondiff'` for nondifferential data encoding. If *`dataenc`* is `'diff'`, `M` must be no greater than 4.

`ber = berawgn(EbNo,'oqpsk',`*`dataenc`*`)` returns the BER of coherently detected offset-QPSK over an uncoded AWGN channel.

`ber = berawgn(EbNo,'dpsk',M)` returns the BER of uncoded DPSK modulation over an AWGN channel.

`ber = berawgn(EbNo,'fsk',M,`*`coherence`*`)` returns the BER of orthogonal uncoded FSK modulation over an AWGN channel. *`coherence`* is either `'coherent'` for coherent demodulation or `'noncoherent'` for noncoherent demodulation. `M` must be no greater than 64 for `'noncoherent'`.

`ber = berawgn(EbNo,'fsk',2,`*`coherence`*`,rho)` returns the BER for binary nonorthogonal FSK over an uncoded AWGN channel, where `rho` is the complex correlation coefficient. See "Nonorthogonal 2-FSK with Coherent Detection" for the definition of the complex correlation coefficient and how to compute it for nonorthogonal BFSK.

`ber = berawgn(EbNo,'msk',`*`precoding`*`)` returns the BER of coherently detected MSK modulation over an uncoded AWGN channel. Setting *`precoding`* to `'off'` returns results for conventional MSK while setting *`precoding`* to `'on'` returns results for precoded MSK.

`ber = berawgn(EbNo,'msk',`*`precoding`*`,coherence)` specifies whether the detection is coherent or noncoherent.

`berlb = berawgn(EbNo,'cpfsk',M,modindex,kmin)` returns a lower bound on the BER of uncoded CPFSK modulation over an AWGN channel. `modindex` is the modulation index, a positive real number. `kmin` is the number of paths having the minimum distance; if this number is unknown, you can assume a value of 1.

`[BER,SER] = berawgn(EbNo, ...)` returns both the BER and SER.

Limitations

The numerical accuracy of this function's output is limited by approximations related to the numerical implementation of the expressions.

You can generally rely on the first couple of significant digits of the function's output.

References

[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg, *Digital Phase Modulation*, New York, Plenum Press, 1986.

[2] Cho, K., and Yoon, D., "On the general BER expression of one- and two-dimensional amplitude modulations", *IEEE Trans. Commun.*, Vol. 50, Number 7, pp. 1074-1080, 2002.

[3] Lee, P. J., "Computation of the bit error rate of coherent M-ary PSK with Gray code bit mapping", *IEEE Trans. Commun.*, Vol. COM-34, Number 5, pp. 488-491, 1986.

[4] Proakis, J. G., *Digital Communications*, 4th ed., McGraw-Hill, 2001.

[5] Simon, M. K, Hinedi, S. M., and Lindsey, W. C., *Digital Communication Techniques – Signal Design and Detection*, Prentice-Hall, 1995.

[6] Simon, M. K, "On the bit-error probability of differentially encoded QPSK and offset QPSK in the presence of carrier synchronization", *IEEE Trans. Commun.*, Vol. 54, pp. 806-812, 2006.

[7] Lindsey, W. C., and Simon, M. K, *Telecommunication Systems Engineering*, Englewood Cliffs, N.J., Prentice-Hall, 1973.

See Also

`bercoding` | `berfading` | `bersync`

How To

- "Theoretical Results"
- Analytical Expressions Used in berawgn

Purpose

Bit error rate (BER) for coded AWGN channels

Syntax

```
berub = bercoding(EbNo,'conv',decision,coderate,dspec)
berub = bercoding(EbNo,'block','hard',n,k,dmin)
berub = bercoding(EbNo,'block','soft',n,k,dmin)
berapprox = bercoding(EbNo,'Hamming','hard',n)
berub = bercoding(EbNo,'Golay','hard',24)
berapprox = bercoding(EbNo,'RS','hard',n,k)
```

Alternatives

As an alternative to the `bercoding` function, invoke the BERTool GUI (`bertool`) and use the **Theoretical** tab.

Description

`berub = bercoding(EbNo,'conv',`*`decision`*`,coderate,dspec)` returns an upper bound or approximation on the BER of a binary convolutional code with coherent phase shift keying (PSK) modulation over an additive white Gaussian noise (AWGN) channel. `EbNo` is the ratio of bit energy to noise power spectral density, in dB. If `EbNo` is a vector, `berub` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels. To specify hard-decision decoding, set *`decision`* to `'hard'`; to specify soft-decision decoding, set *`decision`* to `'soft'`. The convolutional code has code rate equal to `coderate`. The `dspec` input is a structure that contains information about the code's distance spectrum:

- `dspec.dfree` is the minimum free distance of the code.
- `dspec.weight` is the weight spectrum of the code.

To find distance spectra for some sample codes, use the `distspec` function or see [5] and [3].

Note The results for binary PSK and quaternary PSK modulation are the same. This function does not support M-ary PSK when M is other than 2 or 4.

`berub = bercoding(EbNo,'block','hard',n,k,dmin)` returns an upper bound on the BER of an [n,k] binary block code with hard-decision decoding and coherent BPSK or QPSK modulation. `dmin` is the minimum distance of the code.

`berub = bercoding(EbNo,'block','soft',n,k,dmin)` returns an upper bound on the BER of an [n,k] binary block code with soft-decision decoding and coherent BPSK or QPSK modulation. `dmin` is the minimum distance of the code.

`berapprox = bercoding(EbNo,'Hamming','hard',n)` returns an approximation of the BER of a Hamming code using hard-decision decoding and coherent BPSK modulation. (For a Hamming code, if n is known, then k can be computed directly from n.)

`berub = bercoding(EbNo,'Golay','hard',24)` returns an upper bound of the BER of a Golay code using hard-decision decoding and coherent BPSK modulation. Support for Golay currently is only for n=24. In accordance with [3], the Golay coding upper bound assumes only the correction of 3-error patterns. Even though it is theoretically possible to correct approximately 19% of 4-error patterns, most decoders in practice do not have this capability.

`berapprox = bercoding(EbNo,'RS','hard',n,k)` returns an approximation of the BER of (n,k) Reed-Solomon code using hard-decision decoding and coherent BPSK modulation.

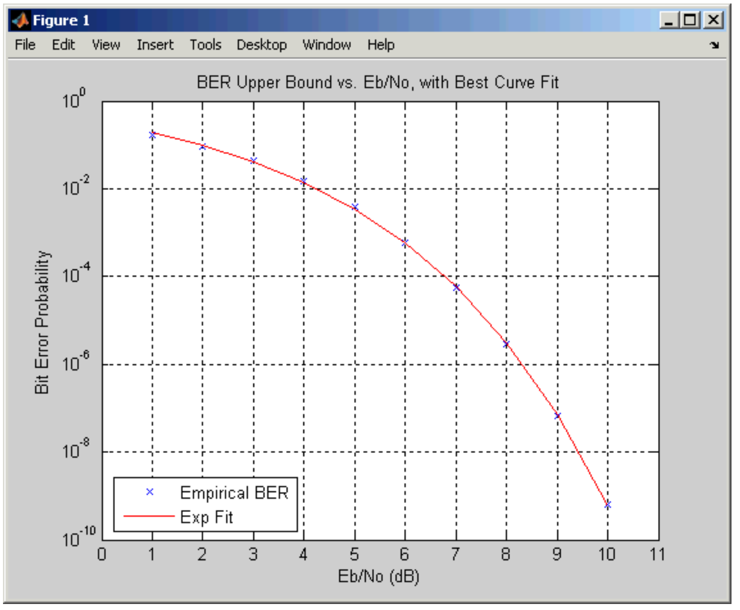
Examples

An example using this function for a convolutional code is in “Plotting Theoretical Error Rates”.

The following example finds an upper bound on the theoretical BER of a block code. It also uses the `berfit` function to perform curve fitting.

```
n = 23; k = 12; % Lengths of codewords and messages
dmin = 7; % Minimum distance
EbNo = 1:10;
ber_block = bercoding(EbNo,'block','hard',n,k,dmin);
berfit(EbNo,ber_block) % Plot BER points and fitted curve.
ylabel('Bit Error Probability');
```

```
title('BER Upper Bound vs. Eb/No, with Best Curve Fit');
```



Limitations

The numerical accuracy of this function's output is limited by

- Approximations in the analysis leading to the closed-form expressions that the function uses
- Approximations related to the numerical implementation of the expressions

You can generally rely on the first couple of significant digits of the function's output.

References

[1] Proakis, J. G., *Digital Communications*, 4th ed., New York, McGraw-Hill, 2001.

[2] Frenger, P., P. Orten, and T. Ottosson, "Convolutional Codes with Optimum Distance Spectrum," *IEEE Communications Letters*, Vol. 3, No. 11, Nov. 1999, pp. 317–319.

[3] Odenwalder, J. P., *Error Control Coding Handbook*, Final Report, LINKABIT Corporation, San Diego, CA, 1976.

[4] Sklar, B., *Digital Communications*, 2nd ed., Prentice Hall, 2001.

[5] Ziemer, R. E., and R. L., Peterson, *Introduction to Digital Communication*, 2nd ed., Prentice Hall, 2001.

See Also

`berawgn` | `berfading` | `bersync` | `distspec`

How To

- "Theoretical Performance Results"
- Analytical Expressions Used in bercoding and BERTool

Purpose

Bit error rate (BER) and confidence interval of Monte Carlo simulation

Syntax

```
[ber,interval] = berconfint(nerrs,ntrials)
[ber,interval] = berconfint(nerrs,ntrials,level)
```

Description

`[ber,interval] = berconfint(nerrs,ntrials)` returns the error probability estimate `ber` and the 95% confidence interval `interval` for a Monte Carlo simulation of `ntrials` trials with `nerrs` errors. `interval` is a two-element vector that lists the endpoints of the interval. If the errors and trials are measured in bits, the error probability is the bit error rate (BER); if the errors and trials are measured in symbols, the error probability is the symbol error rate (SER).

`[ber,interval] = berconfint(nerrs,ntrials,level)` specifies the confidence level as a real number between 0 and 1.

Examples

If a simulation of a communication system results in 100 bit errors in 10^6 trials, the BER (bit error rate) for that simulation is the quotient 10^{-4}. The command below finds the 95% confidence interval for the BER of the system.

```
nerrs = 100; % Number of bit errors in simulation
ntrials = 10^6; % Number of trials in simulation
level = 0.95; % Confidence level
[ber,interval] = berconfint(nerrs,ntrials,level)
```

The output below shows that, with 95% confidence, the BER for the system is between 0.0000814 and 0.0001216.

```
ber =

  1.0000e-004


interval =

  1.0e-003 *
```

```
    0.0814    0.1216
```

For an example that uses the output of `berconfint` to plot error bars on a BER plot, see "Example: Curve Fitting for an Error Rate Plot"

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

See Also

`binofit` | `mle`

Purpose

Bit error rate (BER) for Rayleigh and Rician fading channels

Syntax

```
ber = berfading(EbNo,'pam',M,divorder)
ber = berfading(EbNo,'qam',M,divorder)
ber = berfading(EbNo,'psk',M,divorder)
ber = berfading(EbNo,'depsk',M,divorder)
ber = berfading(EbNo,'oqpsk',divorder)
ber = berfading(EbNo,'dpsk',M,divorder)
ber = berfading(EbNo,'fsk',M,divorder,coherence)
ber = berfading(EbNo,'fsk',2,divorder,coherence,rho)
ber = berfading(EbNo,...,K)
ber = berfading(EbNo,'psk',2,1,K,phaserr)
[BER,SER] = berfading(EbNo, ...)
```

Alternatives

As an alternative to the `berfading` function, invoke the BERTool GUI (`bertool`), and use the **Theoretical** tab.

Description

For All Syntaxes

The first input argument, `EbNo`, is the ratio of bit energy to noise power spectral density, in dB. If `EbNo` is a vector, the output `ber` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels.

Most syntaxes also have an `M` input that specifies the alphabet size for the modulation. `M` must have the form 2^k for some positive integer `k`.

`berfading` uses expressions that assume Gray coding. If you use binary coding, the results may differ.

For cases where diversity is used, the E_b/N_0 on each diversity branch is `EbNo/divorder`, where `divorder` is the diversity order (the number of diversity branches) and is a positive integer.

For Specific Syntaxes

`ber = berfading(EbNo,'pam',M,divorder)` returns the BER for PAM over an uncoded Rayleigh fading channel with coherent demodulation.

`ber = berfading(EbNo,'qam',M,divorder)` returns the BER for QAM over an uncoded Rayleigh fading channel with coherent demodulation.

The alphabet size, `M`, must be at least 4. When $k = \log_2 M$ is odd, a rectangular constellation of size $M = I \times J$ is used, where $I = 2^{\frac{k-1}{2}}$ and $J = 2^{\frac{k+1}{2}}$.

`ber = berfading(EbNo,'psk',M,divorder)` returns the BER for coherently detected PSK over an uncoded Rayleigh fading channel.

`ber = berfading(EbNo,'depsk',M,divorder)` returns the BER for coherently detected PSK with differential data encoding over an uncoded Rayleigh fading channel. Only $M = 2$ is currently supported.

`ber = berfading(EbNo,'oqpsk',divorder)` returns the BER of coherently detected offset-QPSK over an uncoded Rayleigh fading channel.

`ber = berfading(EbNo,'dpsk',M,divorder)` returns the BER for DPSK over an uncoded Rayleigh fading channel. For DPSK, it is assumed that the fading is slow enough that two consecutive symbols are affected by the same fading coefficient.

`ber = berfading(EbNo,'fsk',M,divorder,coherence)` returns the BER for orthogonal FSK over an uncoded Rayleigh fading channel. `coherence` should be `'coherent'` for coherent detection, or `'noncoherent'` for noncoherent detection.

`ber = berfading(EbNo,'fsk',2,divorder,coherence,rho)` returns the BER for binary nonorthogonal FSK over an uncoded Rayleigh fading channel. `rho` is the complex correlation coefficient. See "Nonorthogonal 2-FSK with Coherent Detection" for the definition of the complex correlation coefficient and how to compute it for nonorthogonal BFSK.

`ber = berfading(EbNo,...,K)` returns the BER over an uncoded Rician fading channel, where `K` is the ratio of specular to diffuse energy in linear scale. For the case of `'fsk'`, `rho` must be specified before `K`.

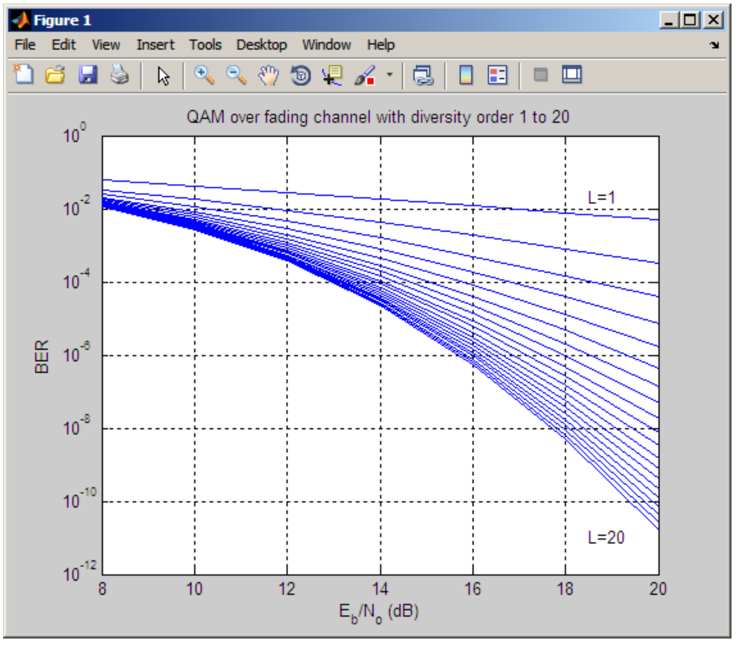
`ber = berfading(EbNo,'psk',2,1,K,phaserr)` returns the BER of BPSK over an uncoded Rician fading channel with imperfect phase synchronization. `phaserr` is the standard deviation of the reference carrier phase error in radians.

`[BER,SER] = berfading(EbNo, ...)` returns both the BER and SER.

Examples

The following example computes and plots the BER for uncoded DQPSK (differential quaternary phase shift keying) modulation over a flat Rayleigh fading channel for several diversity order values.

```
EbNo = 8:2:20;
M = 16; % Use 16 QAM
L = 1; % Start without diversity
ber = berfading(EbNo,'qam',M,L);
semilogy(EbNo,ber);
text(18.5, 0.02, sprintf('L=%d', L))
hold on
% Loop over diversity order, L, 2 to 20
for L=2:20
    ber = berfading(EbNo,'qam',M,L);
    semilogy(EbNo,ber);
end
text(18.5, 1e-11, sprintf('L=%d', L))
title('QAM over fading channel with diversity order 1 to 20')
xlabel('E_b/N_o (dB)')
ylabel('BER')
grid on
```



Limitations

The numerical accuracy of this function's output is limited by approximations related to the numerical implementation of the expressions

You can generally rely on the first couple of significant digits of the function's output.

References

[1] Proakis, John G., *Digital Communications*, 4th ed., New York, McGraw-Hill, 2001.

[2] Modestino, James W., and Mui, Shou Y., *Convolutional code performance in the Rician fading channel*, IEEE Trans. Commun., 1976.

[3] Cho, K., and Yoon, D., "On the general BER expression of one- and two-dimensional amplitude modulations", *IEEE Trans. Commun.*, Vol. 50, Number 7, pp. 1074-1080, 2002.

[4] Lee, P. J., “Computation of the bit error rate of coherent M-ary PSK with Gray code bit mapping”, *IEEE Trans. Commun.*, Vol. COM-34, Number 5, pp. 488-491, 1986.

[5] Lindsey, W. C., “Error probabilities for Rician fading multichannel reception of binary and N-ary signals”, *IEEE Trans. Inform. Theory*, Vol. IT-10, pp. 339-350, 1964.

[6] Simon, M. K , Hinedi, S. M., and Lindsey, W. C., *Digital Communication Techniques – Signal Design and Detection*, Prentice-Hall, 1995.

[7] Simon, M. K., and Alouini, M. S., *Digital Communication over Fading Channels – A Unified Approach to Performance Analysis*, 1st ed., Wiley, 2000.

[8] Simon, M. K , “On the bit-error probability of differentially encoded QPSK and offset QPSK in the presence of carrier synchronization”, *IEEE Trans. Commun.*, Vol. 54, pp. 806-812, 2006.

See Also

`berawgn` | `bercoding` | `bersync`

How To

- “Theoretical Performance Results”
- Analytical Expressions Used in berfading

Purpose

Fit curve to nonsmooth empirical bit error rate (BER) data

Syntax

```
fitber = berfit(empEbNo,empber)
fitber = berfit(empEbNo,empber,fitEbNo)
fitber = berfit(empEbNo,empber,fitEbNo,options)
fitber = berfit(empEbNo,empber,fitEbNo,options,fittype)
[fitber,fitprops] = berfit(...)
berfit(...)
berfit(empEbNo,empber,fitEbNo,options,'all')
```

Description

`fitber = berfit(empEbNo,empber)` fits a curve to the empirical BER data in the vector `empber` and returns a vector of fitted bit error rate (BER) points. The values in `empber` and `fitber` correspond to the E_b/N_0 values, in dB, given by `empEbNo`. The vector `empEbNo` must be in ascending order and must have at least four elements.

Note The `berfit` function is intended for curve fitting or interpolation, *not* extrapolation. Extrapolating BER data beyond an order of magnitude below the smallest empirical BER value is inherently unreliable.

`fitber = berfit(empEbNo,empber,fitEbNo)` fits a curve to the empirical BER data in the vector `empber` corresponding to the E_b/N_0 values, in dB, given by `empEbNo`. The function then evaluates the curve at the E_b/N_0 values, in dB, given by `fitEbNo` and returns the fitted BER points. The length of `fitEbNo` must equal or exceed that of `empEbNo`.

`fitber = berfit(empEbNo,empber,fitEbNo,options)` uses the structure `options` to override the default options used for optimization. These options are the ones used by the `fminsearch` function. You can create the `options` structure using the `optimset` function. Particularly relevant fields are described in the table below.

Field	Description
`options.Display`	Level of display: `'off'` (default) displays no output; `'iter'` displays output at each iteration; `'final'` displays only the final output; `'notify'` displays output only if the function does not converge.
`options.MaxFunEvals`	Maximum number of function evaluations before optimization ceases. The default is 10^4.
`options.MaxIter`	Maximum number of iterations before optimization ceases. The default is 10^4.
`options.TolFun`	Termination tolerance on the closed-form function used to generate the fit. The default is 10^{-4}.
`options.TolX`	Termination tolerance on the coefficient values of the closed-form function used to generate the fit. The default is 10^{-4}.

`fitber = berfit(empEbNo,empber,fitEbNo,options,`*`fittype`*`)` specifies which closed-form function `berfit` uses to fit the empirical data, from the possible fits listed in "Algorithms" on page 1-57 below. *`fittype`* can be `'exp'`, `'exp+const'`, `'polyRatio'`, or `'doubleExp+const'`. To avoid overriding default optimization options, use `options = []`.

`[fitber,fitprops] = berfit(...)` returns the MATLAB structure `fitprops`, which describes the results of the curve fit. Its fields are described in the table below.

Field	Description
`fitprops.fitType`	The closed-form function type used to generate the fit: `'exp'`, `'exp+const'`, `'polyRatio'`, or `'doubleExp+const'`.
`fitprops.coeffs`	The coefficients used to generate the fit. If the function cannot find a valid fit, `fitprops.coeffs` is an empty vector.
`fitprops.sumSqErr`	The sum squared error between the log of the fitted BER points and the log of the empirical BER points.
`fitprops.exitState`	The exit condition of `berfit`: `'The curve fit converged to a solution.'`, `'The maximum number of function evaluations was exceeded.'`, or `'No desirable fit was found'`.
`fitprops.funcCount`	The number of function evaluations used in minimizing the sum squared error function.
`fitprops.iterations`	The number of iterations taken in minimizing the sum squared error function. This is not necessarily equal to the number of function evaluations.

`berfit(...)` plots the empirical and fitted BER data.

`berfit(empEbNo,empber,fitEbNo,options,'all')` plots the empirical and fitted BER data from all the possible fits, listed in the "Algorithms" on page 1-57 below, that return a valid fit. To avoid overriding default options, use `options = []`.

Note A valid fit must be

- real-valued
- monotonically decreasing
- greater than or equal to 0 and less than or equal to 0.5

If a fit does not confirm to this criteria, it is rejected.

Algorithms

The `berfit` function fits the BER data using unconstrained nonlinear optimization via the `fminsearch` function. The closed-form functions that `berfit` considers are listed in the table below, where *x* is the E_b/N_0 in linear terms (*not* dB) and *f* is the estimated BER. These functions were empirically found to provide close fits in a wide variety of situations, including exponentially decaying BERs, linearly varying BERs, and BER curves with error rate floors.

Value of *fittype*	Functional Expression
`'exp'`	$f(x) = \frac{a_1 \exp\{-[(x-a_2)^{a_3}]\}}{a_4}$
`'exp+const'`	$f(x) = \frac{a_1 \exp[-(x-a_2)^{a_3}]}{a_4} + a_5$

Value of *fittype*	Functional Expression
'polyRatio'	$f(x)=\frac{a_1x^2+a_2x+a_3}{x^3+a_4x^2+a_5x+a_6}$
'doubleExp+const'	$\frac{a_1\exp\left[-(x-a_2)^{a_3}\right]}{a_4}+\frac{a_5\exp\left[-(x-a_6)^{a_7}\right]}{a_8}+a_9$

The sum squared error function that `fminsearch` attempts to minimize is

$$F=\sum[\log(\text{empirical BER})-\log(\text{fitted BER})]^2$$

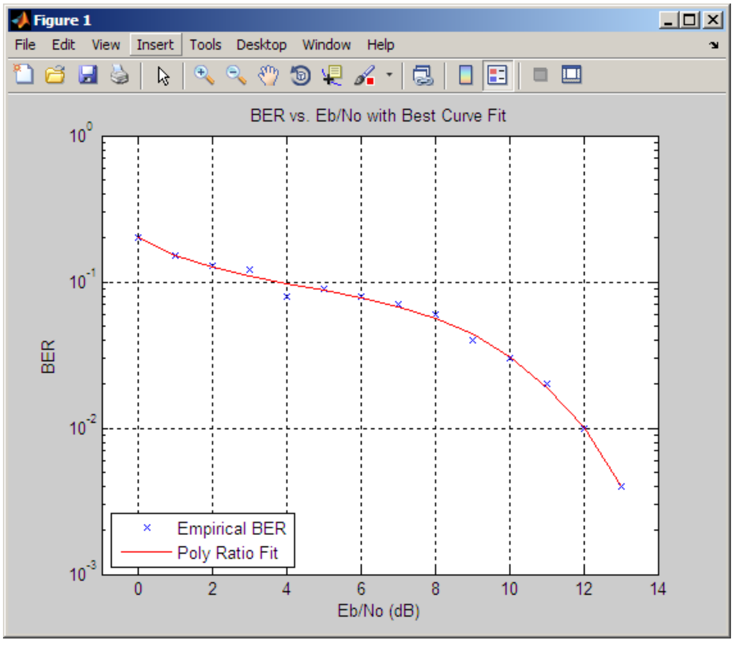
where the fitted BER points are the values in `fitber` and the sum is over the E_b/N_0 points given in `empEbNo`. It is important to use the log of the BER values rather than the BER values themselves so that the high-BER regions do not dominate the objective function inappropriately.

Examples

The examples below illustrate the syntax of the function, but they use hard-coded or theoretical BER data for simplicity. For an example that uses empirical BER data from a simulation, see “Example: Curve Fitting for an Error Rate Plot”.

The code below plots the best fit for a sample set of data.

```
EbNo = 0:13;
berdata = [.2 .15 .13 .12 .08 .09 .08 .07 .06 .04 .03 .02 .01 .004];
berfit(EbNo,berdata); % Plot the best fit.
```

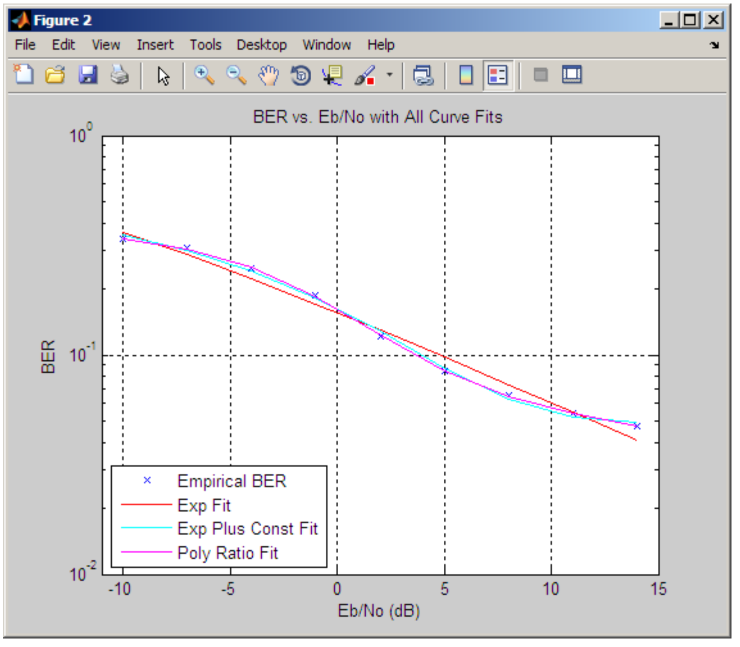


The curve connects the points created by evaluating the fit expression at the values in `EbNo`. To make the curve look smoother, use a syntax like `berfit(EbNo,berdata,[0:0.2:13])`. This alternative syntax uses more points when plotting the curve, but it does not change the fit expression.

The next example demonstrates a fit for a BER curve with an error floor. We generate the empirical BER array by simulating a channel with a null (`ch = [0.5 0.47]`) with BPSK modulation and linear MMSE equalizer at the receiver. We run the berfit with the `'all'` option. The `'doubleExp+const'` fit does not provide a valid fit, and the `'exp'` fit type does not work well for this data. The `'exp+const'` and `'polyRatio'` fits closely match the simulated data.

```
EbNo = -10:3:15;
empBER = [0.3361 0.3076 0.2470 0.1878 0.1212 0.0845 0.0650 0.0540 0.0474];
```

```
figure; berfit(EbNo, empBER, [], [], 'all');
```



The following code illustrates the use of the `options` input structure as well as the `fitprops` output structure. The `'notify'` value for the display level causes the function to produce output when one of the attempted fits does not converge. The `exitState` field of the output structure also indicates which fit converges and which fit does not.

```
M = 4; EbNo = 3:10;
berdata = berfading(EbNo,'psk',M,2); % Compute theoretical BER.
noisydata = berdata.*[.93 .92 1 .59 .08 .15 .01 .01];
% Say when fit fails to converge.
options = optimset('display','notify');

disp('*** Trying exponential fit.') % Poor fit
[fitber1,fitprops1] = berfit(EbNo,noisydata,EbNo,...
```

```
   options,'exp')

disp('*** Trying polynomial ratio fit.') % Good fit
[fitber2,fitprops2] = berfit(EbNo,noisydata,EbNo,...
   options,'polyRatio')
```

The output is as follows:

```
*** Trying exponential fit.

Exiting: Maximum number of function evaluations has been exceeded
         - increase MaxFunEvals option.
         Current function value: 2.729948


fitber1 =

0.0766   0.0423   0.0205   0.0086   0.0030   0.0009   0.0002   0.0000


fitprops1 =

       fitType: 'exp'
        coeffs: [4x1 double]
      sumSqErr: 2.7299
     exitState: 'The maximum number of function evaluations...
                 has been exceeded'
     funcCount: 10000
    iterations: 6177

*** Trying polynomial ratio fit.

fitber2 =

0.0931   0.0476   0.0220   0.0090   0.0031   0.0008   0.0001   0.0001
```

```
fitprops2 =

       fitType: 'polyRatio'
        coeffs: [6x1 double]
      sumSqErr: 2.0578
     exitState: 'The curve fit converged to a solution'
     funcCount: 580
    iterations: 344
```

References

For a general description of unconstrained nonlinear optimization, see the following work.

[1] Chapra, Steven C., and Raymond P. Canale, *Numerical Methods for Engineers*, Fourth Edition, New York, McGraw-Hill, 2002.

See Also

`fminsearch` | `optimset`

Purpose

Bit error rate (BER) for imperfect synchronization

Syntax

```
ber = bersync(EbNo,timerr,'timing')
ber = bersync(EbNo,phaserr,'carrier')
```

Alternatives

As an alternative to the `bersync` function, invoke the BERTool GUI (`bertool`) and use the **Theoretical** tab.

Description

`ber = bersync(EbNo,timerr,'timing')` returns the BER of uncoded coherent binary phase shift keying (BPSK) modulation over an additive white Gaussian noise (AWGN) channel with imperfect timing. The normalized timing error is assumed to have a Gaussian distribution. `EbNo` is the ratio of bit energy to noise power spectral density, in dB. If `EbNo` is a vector, the output `ber` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels. `timerr` is the standard deviation of the timing error, normalized to the symbol interval. `timerr` must be between 0 and 0.5.

`ber = bersync(EbNo,phaserr,'carrier')` returns the BER of uncoded BPSK modulation over an AWGN channel with a noisy phase reference. The phase error is assumed to have a Gaussian distribution. `phaserr` is the standard deviation of the error in the reference carrier phase, in radians.

Examples

The code below computes the BER of coherent BPSK modulation over an AWGN channel with imperfect timing. The example varies both `EbNo` and `timerr`. (When `timerr` assumes the final value of zero, the `bersync` command produces the same result as `berawgn(EbNo,'psk',2)`.)

```
EbNo = [4 8 12];
timerr = [0.2 0.07 0];
ber = zeros(length(timerr), length(EbNo));
for ii = 1:length(timerr)
    ber(ii,:) = bersync(EbNo, timerr(ii),'timerr');
end
% Display result using scientific notation.
format short e; ber
```

```
format; % Switch back to default notation format.
```

The output is below, where each row corresponds to a different value of `timerr` and each column corresponds to a different value of `EbNo`.

```
ber =

  5.2073e-002  2.0536e-002  1.1160e-002
  1.8948e-002  7.9757e-004  4.9008e-006
  1.2501e-002  1.9091e-004  9.0060e-009
```

Limitations

The numerical accuracy of this function's output is limited by

- Approximations in the analysis leading to the closed-form expressions that the function uses
- Approximations related to the numerical implementation of the expressions

You can generally rely on the first couple of significant digits of the function's output.

Limitations Related to Extreme Values of Input Arguments

Inherent limitations in numerical precision force the function to assume perfect synchronization if the value of `timerr` or `phaserr` is very small. The table below indicates how the function behaves under these conditions.

Condition	Behavior of Function
`timerr < eps`	`bersync(EbNo,timerr,'timing')` defined as `berawgn(EbNo,'psk',2)`
`phaserr < eps`	`bersync(EbNo,phaserr,'carrier')` defined as `berawgn(EbNo,'psk',2)`

Algorithms

This function uses formulas from [3].

When the last input is `'timing'`, the function computes

$$\frac{1}{4\pi\sigma}\int_{-\infty}^{\infty}\exp(-\frac{\xi^2}{2\sigma^2})\int_{\sqrt{2R}(1-2|\xi|)}^{\infty}\exp(-\frac{x^2}{2})dxd\xi+\frac{1}{2\sqrt{2\pi}}\int_{\sqrt{2R}}^{\infty}\exp(-\frac{x^2}{2})dx$$

where σ is the `timerr` input and R is the value of `EbNo` converted from dB to a linear scale.

When the last input is `'carrier'`, the function computes

$$\frac{1}{\pi\sigma}\int_{0}^{\infty}\exp(-\frac{\phi^2}{2\sigma^2})\int_{\sqrt{2R}\cos\phi}^{\infty}\exp(-\frac{y^2}{2})dyd\phi$$

where σ is the `phaserr` input and R is the value of `EbNo` converted from dB to a linear scale.

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

[2] Sklar, Bernard, *Digital Communications: Fundamentals and Applications*, Second Edition, Upper Saddle River, NJ, Prentice-Hall, 2001.

[3] Stiffler, J. J., *Theory of Synchronous Communications*, Englewood Cliffs, NJ, Prentice-Hall, 1971.

See Also

`berawgn` | `bercoding` | `berfading`

How To

- “Theoretical Results”

Purpose

Open bit error rate analysis GUI (BERTool)

Syntax

```
bertool
```

Description

`bertool` launches the Bit Error Rate Analysis Tool (BERTool). BERTool is a graphical user interface (GUI) that enables you to analyze BER performance of communications systems. Performance analysis is done via simulation-based, semianalytic, or theoretical approach. See "BERTool" to learn more.

Purpose

Convert binary vectors to decimal numbers

Syntax

```
d = bi2de(b)
d = bi2de(b,flg)
d = bi2de(b,p)
d = bi2de(b,p,flg)
```

Description

`d = bi2de(b)` converts a binary row vector `b` to a nonnegative decimal integer. If `b` is a matrix, each row is interpreted separately as a binary number. In this case, the output `d` is a column vector, each element of which is the decimal representation of the corresponding row of `b`.

Note By default, `bi2de` interprets the first column of `b` as the *lowest*-order digit.

`d = bi2de(b,`*`flg`*`)` is the same as the syntax above, except that *`flg`* is a string that determines whether the first column of `b` contains the lowest-order or highest-order digits. Possible values for *`flg`* are `'right-msb'` and `'left-msb'`. The value `'right-msb'` produces the default behavior.

`d = bi2de(b,p)` converts a base-`p` row vector b to a nonnegative decimal integer , where `p` is an integer greater than or equal to 2. The first column of `b` is the *lowest* base-`p` digit. If `b` is a matrix, the output `d` is a nonnegative decimal vector, each row of which is the decimal form of the corresponding row of `b`.

`d = bi2de(b,p,`*`flg`*`)` is the same as the syntax above, except that *`flg`* is a string that determines whether the first column of `b` contains the lowest-order or highest-order digits. Possible values for *`flg`* are `'right-msb'` and `'left-msb'`. The value `'right-msb'` produces the default behavior.

Examples

Generate a matrix that contains binary representations of five random numbers between 0 and 15, and then convert all five numbers to decimal integers.

```
b = randi([0 1],5,4);   % Generate a 5-by-4 random binary matrix.
de = bi2de(b);
disp('    Dec          Binary')
disp('   -----   -------------------')
disp([de, b])
```

Sample output is below. Your results might vary because the numbers are random.

```
    Dec          Binary
   -----    -------------------
    13      1      0      1      1
     7      1      1      1      0
    15      1      1      1      1
     4      0      0      1      0
     9      1      0      0      1
```

Convert a base-five number into its decimal counterpart, using the leftmost base-five digit (4 in this case) as the most significant digit. This example reflects the fact that $4(5^3) + 2(5^2) + 5^0 = 551$.

```
d = bi2de([4 2 0 1],5,'left-msb')
```

The output is

```
d =

   551
```

See Also

`de2bi`

Purpose

Convert positive integers into corresponding Gray-encoded integers

Syntax

```
y = bin2gray(x,modulation,M)
[y,map] = bin2gray(x,modulation,M)
```

Description

`y = bin2gray(x,modulation,M)` generates a Gray-encoded vector or matrix output `y` with the same dimensions as its input parameter `x`. `x` can be a scalar, vector, or matrix. `modulation` is the modulation type and must be a string equal to `'qam'`, `'pam'`, `'fsk'`, `'dpsk'`, or `'psk'`. `M` is the modulation order that can be an integer power of 2.

`[y,map] = bin2gray(x,modulation,M)` generates a Gray-encoded output `y` with its respective Gray-encoded constellation map, `map`.

You can use map output to label a Gray-encoded constellation. The map output gives the Gray encoded labels for the corresponding modulation. See the example below.

Note If you are converting binary coded data to Gray-coded data and modulating the result immediately afterwards, you should use the appropriate modulation object or function with the `'Gray'` option, instead of BIN2GRAY.

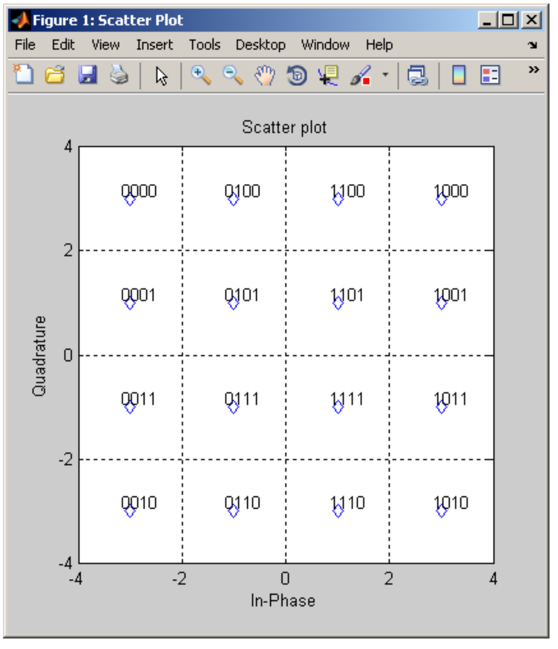
Examples

Gray encode a vector *x* with a 16-QAM Gray encoded constellation and plot its map.

```
% To Gray encode a vector x with a 16-QAM Gray encoded
% constellation and return its map, use:
x=randi([0 15],1,100);
[y,map] = bin2gray(x,'qam',16);
% Obtain the symbols for 16-QAM
hMod = modem.qammod('M', 16);
symbols = hMod.Constellation;
% Plot the constellation
scatterplot(symbols);
set(get(gca,'Children'),'Marker','d','MarkerFaceColor',...
```

```
'auto'); hold on;
% Label the constellation points according
% to the Gray mapping
for jj=1:16
text(real(symbols(jj))-0.15,imag(symbols(jj))+0.15,...
dec2base(map(jj),2,4));
end
set(gca,'yTick',(-4:2:4),'xTick',(-4:2:4),...
'XLim',[-4 4],'YLim',...
[-4 4],'Box','on','YGrid','on', 'XGrid','on');
```

The example code generates the following plot, which shows the 16 QAM constellation with Gray-encoded labeling.



See Also

`gray2bin`

How To

- Gray Encoding a Modulated Signal

biterr

Purpose

Compute number of bit errors and bit error rate (BER)

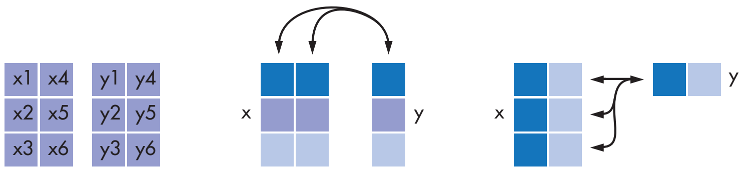
Syntax

```
[number,ratio] = biterr(x,y)
[number,ratio] = biterr(x,y,k)
[number,ratio] = biterr(x,y,k,flg)
[number,ratio,individual] = biterr(...)
```

Description

For All Syntaxes

The `biterr` function compares unsigned binary representations of elements in `x` with those in `y`. The schematics below illustrate how the shapes of `x` and `y` determine which elements `biterr` compares.



(a) Compares x1 with y1, x2 with y2, and so on.

(b) Compares column vector y with each column of matrix x

(c) Compares row vector y with each row of matrix x

Each element of `x` and `y` must be a nonnegative decimal integer; `biterr` converts each element into its natural unsigned binary representation. `number` is a scalar or vector that indicates the number of bits that differ. `ratio` is `number` divided by the *total number of bits*. The total number of bits, the size of `number`, and the elements that `biterr` compares are determined by the dimensions of `x` and `y` and by the optional parameters.

For Specific Syntaxes

`[number,ratio] = biterr(x,y)` compares the elements in `x` and `y`. If the largest among all elements of `x` and `y` has exactly k bits in its simplest binary representation, the total number of bits is k times the number of entries in the *smaller* input. The sizes of `x` and `y` determine which elements are compared:

- If x and y are matrices of the same dimensions, then biterr compares x and y element by element. number is a scalar. See schematic (a) in the preceding figure.
- If one is a row (respectively, column) vector and the other is a two-dimensional matrix, then biterr compares the vector element by element with *each row (resp., column)* of the matrix. The length of the vector must equal the number of columns (resp., rows) in the matrix. number is a column (resp., row) vector whose mth entry indicates the number of bits that differ when comparing the vector with the mth row (resp., column) of the matrix. See schematics (b) and (c) in the figure.

[number,ratio] = biterr(x,y,k) is the same as the first syntax, except that it considers each entry in x and y to have k bits. The total number of bits is k times the number of entries of the smaller of x and y. An error occurs if the binary representation of an element of x or y would require more than k digits.

[number,ratio] = biterr(x,y,k,*flg*) is similar to the previous syntaxes, except that *flg* can override the defaults that govern which elements biterr compares and how biterr computes the outputs. The possible values of *flg* are 'row-wise', 'column-wise', and 'overall'. The table below describes the differences that result from various combinations of inputs. As always, ratio is number divided by the total number of bits. If you do not provide k as an input argument, the function defines it internally as the number of bits in the simplest binary representation of the largest among all elements of x and y.

Comparing a Two-Dimensional Matrix x with Another Input y

Shape of y	flg	Type of Comparison	number	Total Number of Bits
2-D matrix	`'overall'` (default)	Element by element	Total number of bit errors	`k` times number of entries of `y`
	`'row-wise'`	mth row of `x` vs. mth row of `y`	Column vector whose entries count bit errors in each row	`k` times number of entries of `y`
	`'column-wise'`	mth column of `x` vs. mth column of `y`	Row vector whose entries count bit errors in each column	`k` times number of entries of `y`

Comparing a Two-Dimensional Matrix x with Another Input y (Continued)

Shape of y	flg	Type of Comparison	number	Total Number of Bits
Row vector	'overall'	y vs. each row of x	Total number of bit errors	k times number of entries of x
	'row-wise' (default)	y vs. each row of x	Column vector whose entries count bit errors in each row of x	k times size of y
Column vector	'overall'	y vs. each column of x	Total number of bit errors	k times number of entries of x
	'column-wise' (default)	y vs. each column of x	Row vector whose entries count bit errors in each column of x	k times size of y

`[number,ratio,individual] = biterr(...)` returns a matrix `individual` whose dimensions are those of the larger of `x` and `y`. Each entry of `individual` corresponds to a comparison between a pair of elements of `x` and `y`, and specifies the number of bits by which the elements in the pair differ.

Examples

Example 1

The commands below compare the column vector [0; 0; 0] to each column of a random binary matrix. The output is the number, proportion, and locations of 1s in the matrix. In this case, `individual` is the same as the random matrix.

```
format rat;
[number,ratio,individual] = biterr([0;0;0],randi([0 1],3,5))
```

The output is

```
number =

     2          0          0          3          1


ratio =

   2/3          0          0          1        1/3


individual =

     1          0          0          1          0
     1          0          0          1          0
     0          0          0          1          1
```

Example 2

The commands below illustrate the use of *`flg`* to override the default row-by-row comparison. `number` and `ratio` are scalars, and `individual` has the same dimensions as the larger of the first two arguments of `biterr`.

```
format rat;
[number2,ratio2,individual2] = biterr([1 2; 3 4],[1 3],3,'overall')
```

The output is

```
number2 =

      5


ratio2 =

     5/12


individual2 =

      0             1
      1             3
```

Example 3

The script below adds errors to 10% of the elements in a matrix. Each entry in the matrix is a two-bit number in decimal form. The script computes the bit error rate using `biterr` and the symbol error rate using `symerr`.

```
x = randi([0 3],100); % Original signal
% Create errors to add to ten percent of the elements of x.
% Errors can be either 1, 2, or 3 (not zero).
errorplace = (rand(100,100) > .9); % Where to put errors
errorvalue = randi(3,100); % Value of the errors
errors = errorplace.*errorvalue;
y = rem(x+errors,4); % Signal with errors added, mod 4
format short
[num_bit,ratio_bit] = biterr(x,y,2)
[num_sym,ratio_sym] = symerr(x,y)
```

Sample output is below. `ratio_sym` is close to the target value of `0.10`. Your results might vary because the example uses random numbers.

```
num_bit =

        1304


ratio_bit =

    0.0652


num_sym =

   981


ratio_sym =

    0.0981
```

Example 4

The following example uses logical input arguments.

```
SNR = 3; frameLen = 100;
x = randi([0 1], frameLen, 1);
y = awgn(2*x-1, SNR);
z = y > 0;
biterr(x, z)
```

Example 5

The following example uses logical input arguments.

```
SNR = 5; frameLen = 100;
x = rand(100, 1) > 0.5;
y = awgn(2*x-1, SNR);
```

```
z = y > 0;
biterr(x, z)
```

See Also

`symerr`

Purpose

Model binary symmetric channel

Syntax

```
ndata = bsc(data,p)
ndata = bsc(data,p,s)
ndata = bsc(data,p,state)
[ndata,err] = bsc(...)
```

Description

`ndata = bsc(data,p)` passes the binary input signal `data` through a binary symmetric channel with error probability `p`. The channel introduces a bit error with probability `p`, processing each element of `data` independently. `data` must be an array of binary numbers or a Galois array in GF(2). `p` must be a scalar between 0 and 1.`ndata = bsc(data,p,s)` causes rand to use the random stream s. s is any valid random stream. See RandStream for more details.

`ndata = bsc(data,p,state)` resets the state of the uniform random number generator `rand` to the integer `state`.

Note This usage is deprecated and may be removed in a future release. Instead of `state`, use `s`, as in the previous example.

`[ndata,err] = bsc(...)` returns an array, `err`, containing the channel errors.

This function uses, by default, the Mersenne Twister algorithm by Nishimura and Matsumoto.

Note Using the `state` parameter causes this function to switch random generators to use the `'state'` algorithm of the `rand` function.

See `rand` for details on the generator algorithm.

Examples

To introduce bit errors in the bits in a random matrix with probability 0.15, use the `bsc` function:

```
z = randi([0 1],100,100); % Random matrix
nz = bsc(z,.15); % Binary symmetric channel
[numerrs, pcterrs] = biterr(z,nz) % Number and percentage of errors
```

The output below is typical. The percentage of bit errors is not exactly 15% in most trials, but it is close to 15% if the size of the matrix `z` is large.

```
numerrs =

        1509


pcterrs =

    0.1509
```

Another example using this function is in "Design a Rate 2/3 Feedforward Encoder Using Simulink®".

See Also

`rand` | `awgn`

How To

- "Binary Symmetric Channels"

Purpose

Construct constant modulus algorithm (CMA) object

Syntax

```
alg = cma(stepsize)
alg = cma(stepsize,leakagefactor)
```

Description

The `cma` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Adaptive Algorithms”.

Note After you use either `lineareq` or `dfe` to create a CMA equalizer object, you should initialize the equalizer object’s `Weights` property with a nonzero vector. Typically, CMA is used with differential modulation; otherwise, the initial weights are very important. A typical vector of initial weights has a 1 corresponding to the center tap and 0s elsewhere.

`alg = cma(stepsize)` constructs an adaptive algorithm object based on the constant modulus algorithm (CMA) with a step size of `stepsize`.

`alg = cma(stepsize,leakagefactor)` sets the leakage factor of the CMA. `leakagefactor` must be between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, while a value of 0 corresponds to a memoryless update algorithm.

Properties

The table below describes the properties of the CMA adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see “Access Properties of an Adaptive Algorithm”.

Property	Description
`AlgType`	Fixed value, `'Constant Modulus'`
`StepSize`	CMA step size parameter, a nonnegative real number
`LeakageFactor`	CMA leakage factor, a real number between 0 and 1

Algorithms

Referring to the schematics in “Equalizer Structure”, define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of weights, w, this adaptive algorithm creates the new set of weights given by

(`LeakageFactor`) w + (`StepSize`) u^*e

where the * operator denotes the complex conjugate.

References

[1] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, NJ, Prentice-Hall, 1996.

[2] Johnson, Richard C., Jr., Philip Schniter, Thomas. J. Endres, et al., “Blind Equalization Using the Constant Modulus Criterion: A Review,” *Proceedings of the IEEE*, Vol. 86, October 1998, pp. 1927–1950.

See Also

`lms` | `signlms` | `normlms` | `varlms` | `rls` | `lineareq` | `dfe` | `equalize`

How To

- “Equalization”

comm_links

Purpose

Library link information for Communications System Toolbox blocks

Syntax

```
comm_links
comm_links(sys)
comm_links(sys,color)
```

Description

`comm_links` returns a structure with two elements. Each element contains a cell array of strings containing names of library blocks in the current system. The blocks are grouped into two categories: obsolete and current. Blocks at all levels of the model are analyzed.

`comm_links(sys)` works as above on the named system `sys`, instead of the current system.

`comm_links(sys,`*`color`*`)` additionally colors all obsolete blocks according to the specified *`color`*. *`color`* is one of the following strings: 'blue', 'green', 'red', 'cyan', 'magenta', 'yellow', or 'black'.

Obsolete blocks are blocks that are no longer supported. They might or might not work properly.

Current blocks are supported and represent the latest block functionality.

See Also

liblinks | `commliblist`

Purpose

Open main Communications System Toolbox block library

Syntax

```
commlib
```

Description

`commlib` opens the latest version of the Communications System Toolbox™ block library.

See Also

dsplib

commscope

Purpose

Package of communications scope classes

Syntax

```
h = commscope.<type>(...)
```

Description

`h = commscope.<type>(...)` returns a communications scope object `h` of type `type`.

Type `help commscope` to get a complete list of available types.

Each `type` of communications scope object is equipped with functions for simulation and visualization. Type `help commscope.<type>` to get the complete help on a specific communications scope object (e.g., `help commscope.eyediagram`).

See Also

`commscope.eyediagram`

Purpose

Eye diagram analysis

Syntax

```
h = commscope.eyediagram
h = commscope.eyediagram(property1,value1,...)
```

Description

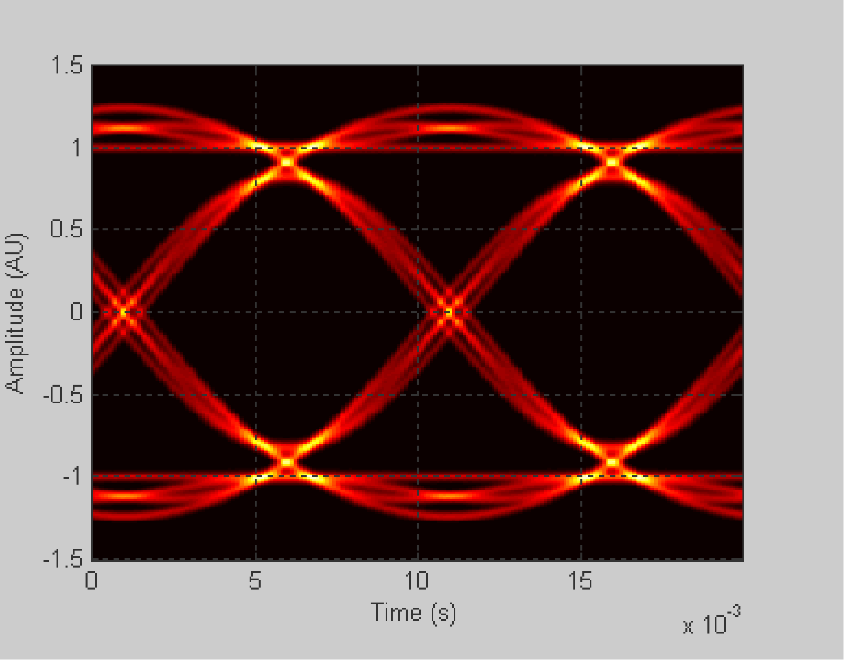
`h = commscope.eyediagram` constructs an eye diagram object, `h`, with default properties. This syntax is equivalent to:

```
H = commscope.eyediagram('SamplingFrequency', 10000, ...
                         'SamplesPerSymbol', 100, ...
                         'SymbolsPerTrace', 2, ...
                         'MinimumAmplitude', -1, ...
                         'MaximumAmplitude', 1, ...
                         'AmplitudeResolution', 0.0100, ...
                         'MeasurementDelay', 0, ...
                         'PlotType', '2D Color', ...
                         'PlotTimeOffset', 0, ...
                         'PlotPDFRange', [0 1], ...
                         'ColorScale', 'linear', ...
                         'RefreshPlot', 'on');
```

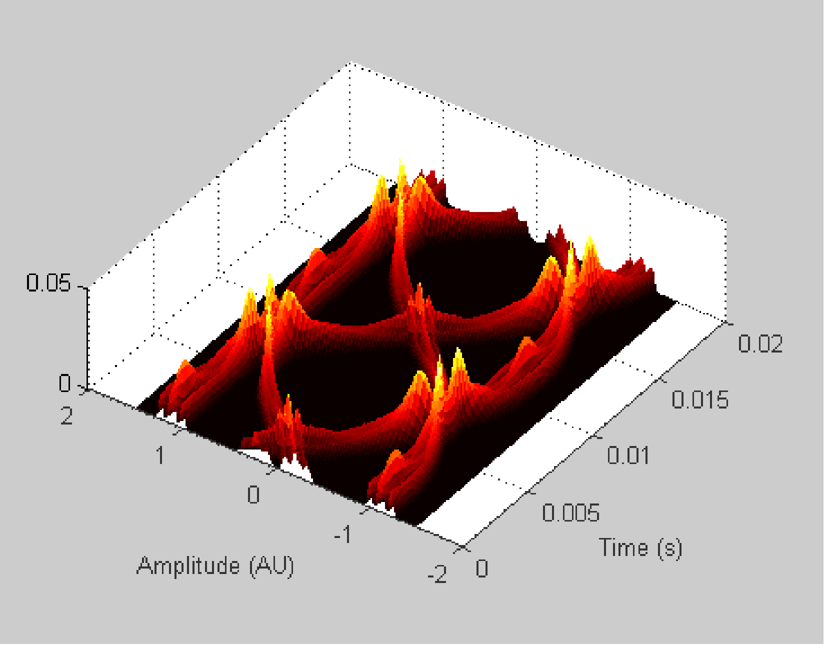
`h = commscope.eyediagram(property1,value1,...)` constructs an eye diagram object, `h`, with properties as specified by property/value pairs.

The eye diagram object creates a series of vertical histograms from zero to T seconds, at T_s second intervals, where T is a multiple of the symbol duration of the input signal and T_s is the sampling time. A vertical histogram is defined as the histogram of the amplitude of the input signal at a given time. The histogram information is used to obtain an approximation to the probability density function (PDF) of the input amplitude distribution. The histogram data is used to generate `'2D Color'` plots, where the color indicates the value of the PDF, and `'3D Color'` plots. The `'2D Line'` plot is obtained by constructing an eye diagram from the last n traces stored in the object, where a trace is defined as the segment of the input signal for a T second interval.

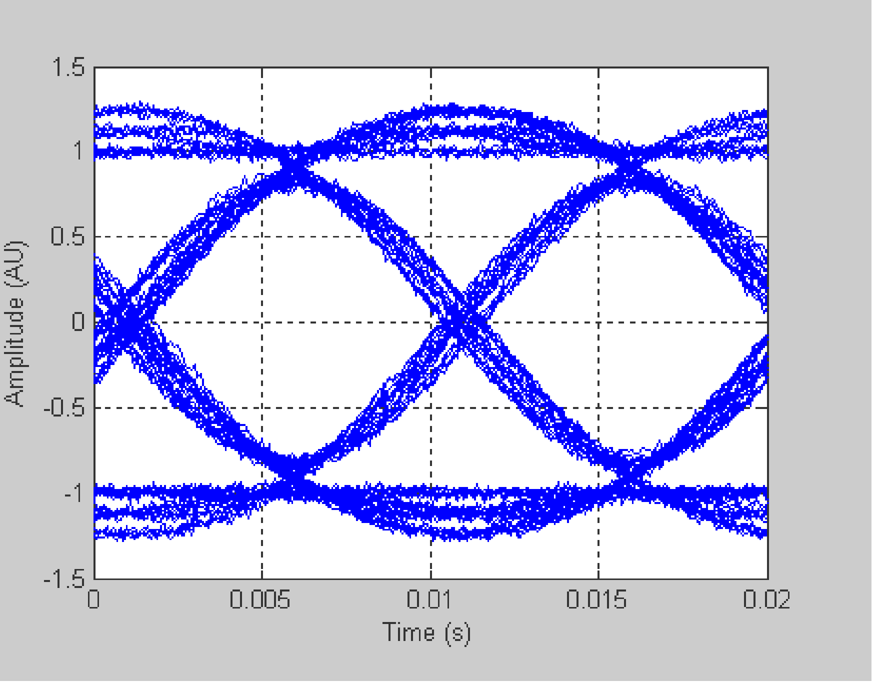
You can change the plot type by setting the `PlotType` property. The following plots are examples of each type.



2D-Color Eye Diagram



3D-Color Eye Diagram



2D-Line Eye Diagram

To see a detailed demonstration of this object's use, type `showdemo scattereyedemo;` at the command line.

Properties

An eye diagram scope object has the properties shown on the following table. All properties are writable except for the ones explicitly noted otherwise.

Property	Description
`Type`	Type of scope object (`'Eye Diagram'`). This property is not writable.
`SamplingFrequency`	Sampling frequency of the input signal in hertz.

Property	Description
SamplesPerSymbol	Number of samples used to represent a symbol. An increase in SamplesPerSymbol improves the resolution of an eye diagram.
SymbolRate	The symbol rate of the input signal. This property is not writable and is automatically computed based on SamplingFrequency and SamplesPerSymbol.
SymbolsPerTrace	The number of symbols spanned on the time axis of the eye diagram scope.
MinimumAmplitude	Minimum amplitude of the input signal. Signal values less than this value are ignored both for plotting and for measurement computation.
MaximumAmplitude	Maximum amplitude of the input signal. Signal values greater than this value are ignored both for plotting and for measurement computation.
AmplitudeResolution	The resolution of the amplitude axis. The amplitude axis is created from MinimumAmplitude to MaximumAmplitude with AmplitudeResolution steps.
MeasurementDelay	The time in seconds the scope waits before starting to collect data.

Property	Description
`PlotType`	Type of the eye diagram plot. The choices are `'2D Color'` (two dimensional eye diagram, where color intensity represents the probability density function values), `'3D Color'` (three dimensional eye diagram, where the z-axis represents the probability density function values), and `'2D Line'` (two dimensional eye diagram, where each trace is represented by a line).
`NumberOfStoredTraces`	The number of traces stored to display the eye diagram in `'2D Line'` mode.
`PlotTimeOffset`	The plot time offset input values must reside in the closed interval [-Tsym Tsym], where Tsym is the symbol duration. Since the eye diagram is periodic, if the value you enter is out of range, it wraps to a position on the eye diagram that is within range.
`RefreshPlot`	The switch that controls the plot refresh style. The choices are `'on'` (the eye diagram plot is refreshed every time the update method is called) and `'off'` (the eye diagram plot is not refreshed when the `update` method is called).
`PlotPDFRange`	The range of the PDF values that will be displayed in the `'2D Color'` mode. The PDF values outside the range are set to a constant mask color.
`ColorScale`	The scale used to represent the color, the z-axis, or both. The choices are `'linear'` (linear scale) and `'log'` (base ten logarithmic scale).

Property	Description
SamplesProcessed	The number of samples processed by the eye diagram object. This value does not include the discarded samples during the MeasurementDelay period. This property is not writable.
OperationMode	When the operation mode is complex signal, the eye diagram collects and plots data on both the in-phase component and the quadrature component. When the operation mode is real signal, the eye diagram collects and plots real signal data.
Measurements	An eye diagram can display various types of measurements. All measurements are done on both the in-phase and quadrature signal, unless otherwise stated. For more information, see the Measurements section.

The resolution of the eye diagram in '2D Color' and '3D Color' modes can be increased by increasing SamplingFrequency, decreasing AmplitudeResolution, or both.

Changing MinimumAmplitude, MaximumAmplitude, AmplitudeResolution, SamplesPerSymbol, SymbolsPerTrace, and MeasurementDelay resets the measurements and updates the eye diagram.

Methods

An eye diagram object is equipped with seven methods for inspection, object management, and visualization.

update

This method updates the eye diagram object data.

update(h,x) updates the collected data of the eye diagram object h with the input x.

If the RefreshPlot property is set to 'on', the update method also refreshes the eye diagram figure.

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram('RefreshPlot', 'off')

% Prepare a noisy sinusoidal as input
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
  'SNR', 20);
x = step(hChan,0.5*sin(2*pi*(0:1/100:10))+j*cos(2*pi*(0:1/100:10)));
% update the eyediagram
update(h, x);
% Check the number of proccessed samples
h.SamplesProcessed
```

plot

This method displays the eye diagram figure.

The plot method has three usage cases:

plot(h) plots the eye diagram for the eye diagram object h with the current colormap or the default linespec.

plot(h,cmap), when used with the plottype set to '2D Color' or '3D Color', plots the eye diagram for the object h, and sets the colormap to cmap.

plot(h,linespec), when used with the plottype set to '2D Line', plots the eye diagram for the object h using linespec as the line specification. See the help for plot for valid linespecs.

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram;
% Prepare a noisy sinusoid as input
```

```
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',
  'SNR', 20);
x = step(hChan, 0.5*sin(2*pi*(0:1/100:10))+ j*0.5*cos(2*pi*(0:1/100:10
% Update the eye diagram
update(h, x);
% Display the eye diagram figure
plot(h)

% Display the eye diagram figure with jet colormap
plot(h, jet(64))

% Display 2D Line eye diagram with red dashed lines
h.PlotType = '2D Line';
plot(h, 'r--')
```

exportdata

This method exports the eye diagram data.

`[VERHIST EYEL HORHISTX HORHISTRF] = EXPORTDATA(H)` Exports the eye diagram data collected by the eyediagram object *H*.

`VERHIST` is a matrix that holds the vertical histogram, which is also used to plot '2D Color' and '3D Color' eye diagrams.

`EYEL` is a matrix that holds the data used to plot 2D Line eye diagram. Each row of the EYEC holds one trace of the input signal.

`HORHISTX` is a matrix that holds the crossing point histogram data collected for the values defined by the CrossingAmplitudes property of the MeasurementSetup object. HORHISTX(i, :) represents the histogram for CrossingAmplitudes(i).

`HORHISTRF` is a matrix that holds the crossing point histograms for rise and fall time levels. HORHISTRF(i,:) represents the histogram for AmplitudeThreshold(i).

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram('RefreshPlot', 'off');
```

```
% Prepare a noisy sinusoidal as input
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
  'SNR', 20);
x = step(hChan, 0.5*sin(2*pi*(0:1/100:10))+ j*0.5*cos(2*pi*(0:1/100:10)));
% Update the eyediagram
update(h, x);
% Export the data
[eyec eyel horhistx horhistrf] = exportdata(h);
% Plot line data
t=0:1/h.SamplingFrequency:h.SymbolsPerTrace/h.SymbolRate;
plot(t, real(eyel)); xlabel('time (s)');...
    ylabel('Amplitude (AU)'); grid on;
% Plot 2D Color data
t=0:1/h.SamplingFrequency:h.SymbolsPerTrace/h.SymbolRate;
a=h.MinimumAmplitude:h.AmplitudeResolution:h.MaximumAmplitude;
imagesc(t,a,eyec); xlabel('time (s)'); ylabel('Amplitude (AU)');
```

reset

This method resets the eye diagram object.

`reset(h)` resets the eye diagram object `h`. Resetting `h` clears all the collected data.

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram('RefreshPlot', 'off');
% Prepare a noisy sinusoidal as input
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
  'SNR', 20);
x = step(hChan, 0.5*sin(2*pi*(0:1/100:10))+ j*0.5*cos(2*pi*(0:1/100:10)));
update(h, x);             % update the eyediagram
h.SamplesProcessed        % Check the number of proccessed samples
reset(h);                 % reset the object
h.SamplesProcessed        % Check the number of proccessed samples
```

copy

This method copies the eye diagram object.

`h = copy(ref_obj)` creates a new eye diagram object `h` and copies the properties of object `h` from properties of `ref_obj`.

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram('MinimumAmplitude', -3, ...
    'MaximumAmplitude', 3);
disp(h); % display object properties
h1 = copy(h)
```

disp

This method displays properties of the eye diagram object.

`disp(h)` displays relevant properties of eye diagram object `h`.

If a property is not relevant to the object's configuration, it is not displayed. For example, for a `commscope.eyediagram` object, the `ColorScale` property is not relevant when `PlotType` property is set to `'2D Line'`. In this case the `ColorScale` property is not displayed.

The following is an example of its use:

```
% Create an eye diagram scope object
h = commscope.eyediagram;
% Display object properties
disp(h);
h = commscope.eyediagram('PlotType', '2D Line')
```

close

This method closes the eye diagram object figure.

`close(h)` closes the figure of the eye diagram object `h`.

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram;
% Call the plot method to display the scope
plot(h);
% Wait for 1 seconds
```

```
pause(1)
% Close the scope
close(h)
```

analyze

This methods executes eye diagram measurements. analyze(h) executes the eye diagram measurements on the collected data of the eye diagram scope object *h*. The results of the measurements are stored in the Measurements property of *h*. See "Measurements" on page 1-98 for more information.

In some cases, the analyze method cannot determine a measurement value. If this problem occurs, verify that your settings for measurement setup values or the eye diagram are valid.

Measurements

You can obtain the following measurements on an eye diagram:

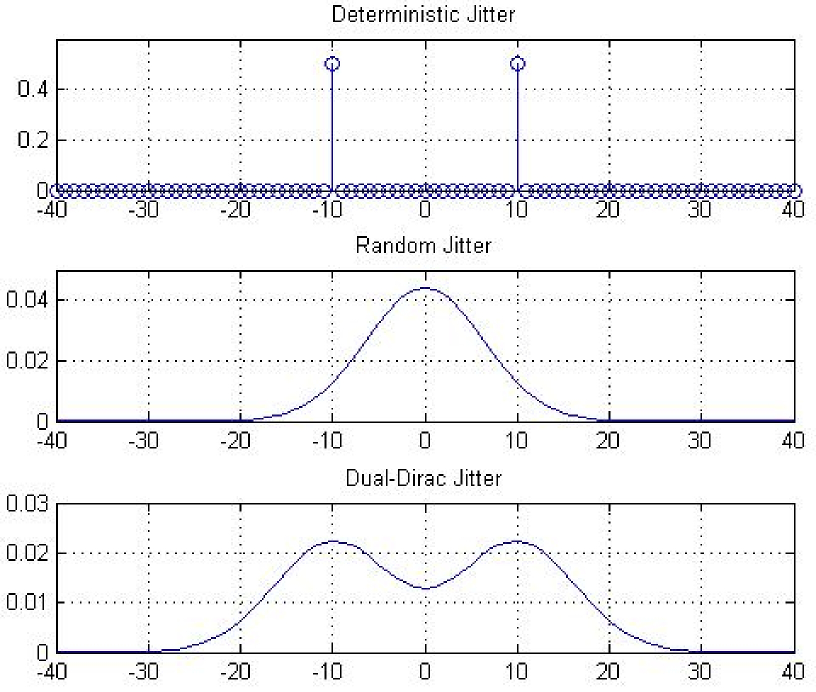
- Amplitude Measurements
  - Eye Amplitude
  - Eye Crossing Amplitude
  - Eye Crossing Percentage
  - Eye Height
  - Eye Level
  - Eye SNR
  - Quality Factor
  - Vertical Eye Opening
- Time Measurements
  - Deterministic Jitter
  - Eye Crossing Time
  - Eye Delay
  - Eye Fall Time

- Eye Rise Time
- Eye Width
- Horizontal Eye Opening
- Peak-to-Peak Jitter
- Random Jitter
- RMS Jitter
- Total Jitter

Measurements assume that the eye diagram object has valid data. A valid eye diagram has two distinct eye crossing points and two distinct eye levels.

The deterministic jitter, horizontal eye opening, quality factor, random jitter, and vertical eye opening measurements utilize a dual-Driac algorithm. *Jitter* is the deviation of a signal's timing event from its intended (ideal) occurrence in time [1]. Jitter can be represented with a dual-Driac model. A dual-Driac model assumes that the jitter has two components: deterministic jitter (DJ) and random jitter (RJ). The DJ PDF comprises two delta functions, one at μ_L and one at μ_R. The RJ PDF is assumed to be Gaussian with zero mean and variance σ.

The *Total Jitter (TJ) PDF* is the convolution of these two PDFs, which is composed of two Gaussian curves with variance σ and mean values μ_L and μ_R. See the following figure.



The dual-Dirac model is described in [5] in more detail. The amplitude of the two Dirac functions may not be the same. In such a case, the analyze method estimates these amplitudes, ρ_L and ρ_R.

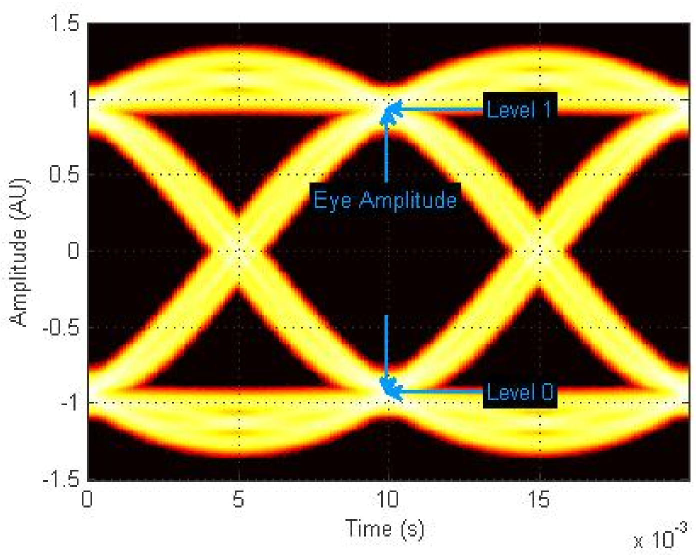
Amplitude Measurements

You can use the vertical histogram to obtain a variety of amplitude measurements. For complex signals, measurements are done on both in-phase and the quadrature components, unless otherwise specified.

Note For amplitude measurements, at least one bin per vertical histogram must reach 10 hits before the measurement is taken, ensuring higher accuracy.

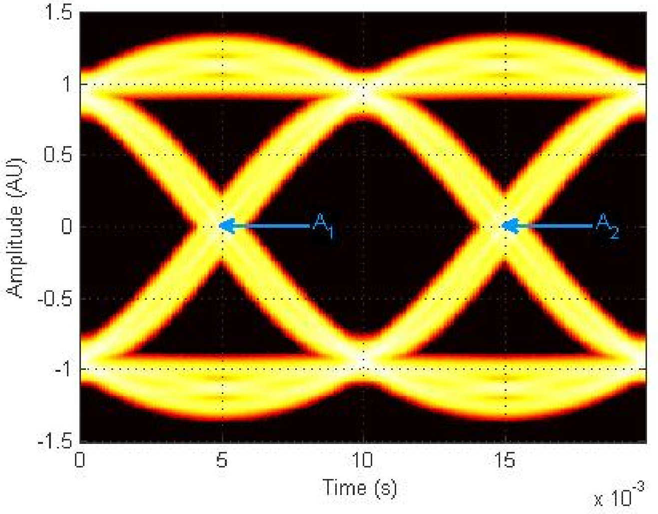
Eye Amplitude (EyeAmplitude)

Eye Amplitude, measured in Amplitude Units (AU), is defined as the distance between two neighboring eye levels. For an NRZ signal, there are only two levels: the high level (level 1 in figure) and the low level (level 0 in figure). The eye amplitude is the difference of these two values, as shown in figure [3].

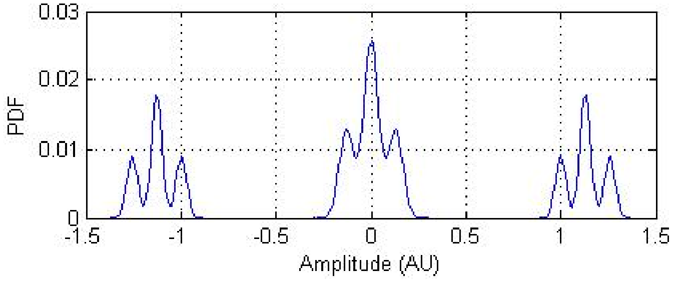


Eye Crossing Amplitude (EyeCrossingLevel)

Eye crossing amplitudes are the amplitude levels at which the eye crossings occur, measured in Amplitude Units (AU). The analyze method calculates this value using the mean value of the vertical histogram at the crossing times [3]. See the following figure.



The next figure shows the vertical histogram at the first eye crossing time.



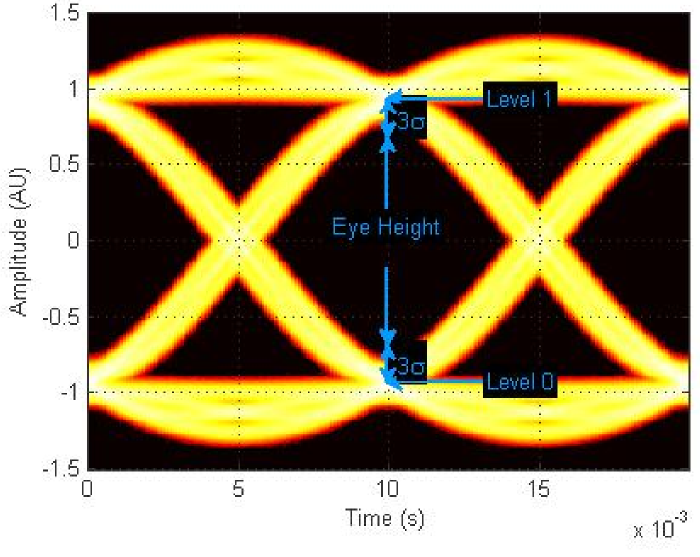
Eye Crossing Percentage (EyeOpeningVer)

Eye Crossing Percentage is the location of the eye crossing levels as a percentage of the eye amplitude.

Eye Height (EyeHeight)

Eye Height, measured in Amplitude Units (AU), is defined as the 3σ distance between two neighboring eye levels.

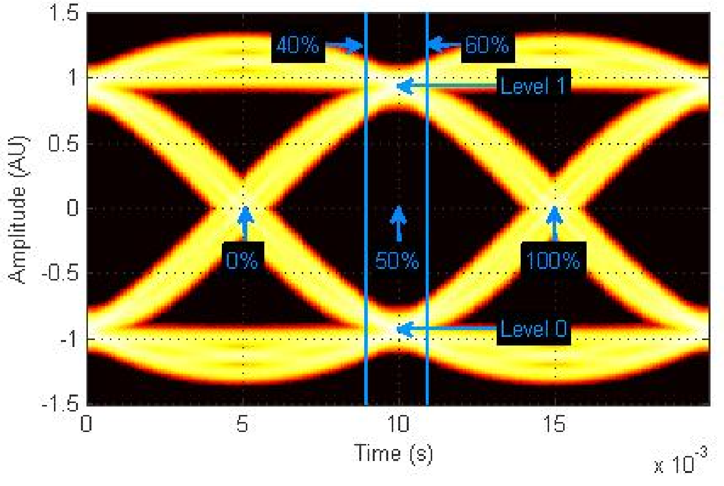
For an NRZ signal, there are only two levels: the high level (level 1 in figure) and the low level (level 0 in figure). The eye height is the difference of the two 3σ points, as shown in the next figure. The 3σ point is defined as the point that is three standard deviations away from the mean value of a PDF.



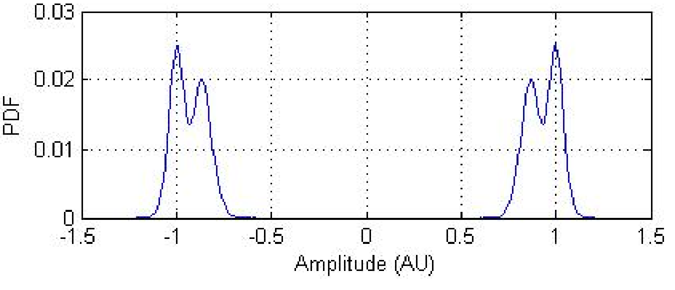
Eye Level (EyeLevel)

Eye Level is the amplitude level used to represent data bits, measured in Amplitude Units (AU).

For an ideal NRZ signal, there are two eye levels: +A and –A. The analyze method calculates eye levels by estimating the mean value of the vertical histogram in a window around the EyeDelay, which is also the 50% point between eye crossing times [3]. The width of this window is determined by the EyeLevelBoundary property of the eyemeasurementsetup object, shown in the next figure.



The analyze method calculates the mean value of all the vertical histograms within the eye level boundaries. The mean vertical histogram appears in the following figure. There are two distinct PDFs, one for each eye level. The mean values of the individual histograms are the eye levels as shown in this figure.



Eye SNR (EyeSNR)

Eye signal-to-noise ratio is defined as the ratio of the eye amplitude to the sum of the standard deviations of the two eye levels. It can be expressed as:

$$\text{SNR} = \frac{L_1 - L_0}{\sigma_1 + \sigma_0}$$

where L_1 and L_0 represent eye level 1 and 0, respectively, and σ_1 and σ_2 are the standard deviation of eye level 1 and 0, respectively.

For an NRZ signal, eye level 1 corresponds to the high level, and the eye level 0 corresponds to low level.

Quality Factor (QualityFactor)

The analyze method calculates *Quality Factor* the same way as the eye SNR. However, instead of using the mean and standard deviation values of the vertical histogram for L_1 and σ_1, the analyze method uses the mean and standard deviation values estimated using the dual-Dirac method. [2] See dual-Dirac section for more detail.

Vertical Eye Opening (EyeOpeningVer)

Vertial Eye Opening is defined as the vertical distance between two points on the vertical histogram at EyeDelay that corresponds to the BER value defined by the BERThreshold property of the eyemeasurementsetup object. The analyze method calculates this measurement taking into account the random and deterministic components using a dual-Dirac model [5] (see the Dual Dirac Section). A typical BER value for the eye opening measurements is 10^{-12}, which approximately corresponds to the 7σ point assuming a Gaussian distribution.

Time Measurements

You can use the horizontal histogram of an eye diagram to obtain a variety of timing measurements.

Note For time measurements, at least one bin per horizontal histogram must reach 10 hits before the measurement is taken.

Deterministic Jitter (JitterDeterministic)

Deterministic Jitter is the deterministic component of the jitter. You calculate it using the tail mean value, which is estimated using the dual-Dirac method as follows [5]:

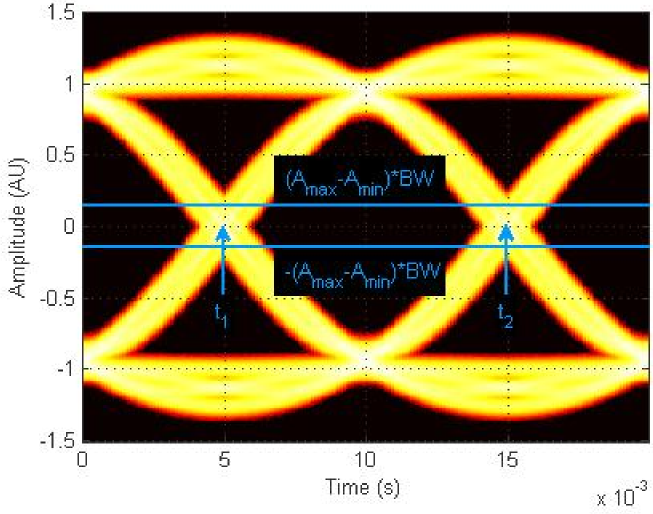
$$DJ = \mu_L - \mu_R$$

where μ_L and μ_R are the mean values returned by the dual-Dirac algorithm.

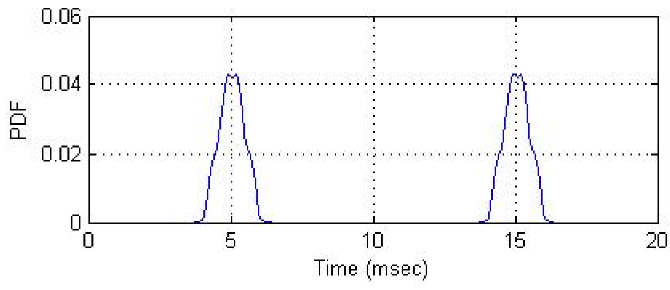
Eye Crossing Time (EyeCrossingTime)

Eye crossing times are calculated as the mean of the horizontal histogram for each crossing point, around the reference amplitude level. This value is measured in seconds. The mean value of all the horizontal PDFs is calculated in a region defined by the CrossingBandWith property of the eyemeasurementsetup object.

The region is from $-A_{total} * BW$ to $+A_{total} * BW$, where A_{total} is the total amplitude range of the eye diagram (i.e., $A_{total} = A_{max} - A_{min}$) and BW is the crossing band width, shown in the following figure.



The following figure shows the average PDF in this region. Because this example assumes two symbols per trace, there are two crossing points.

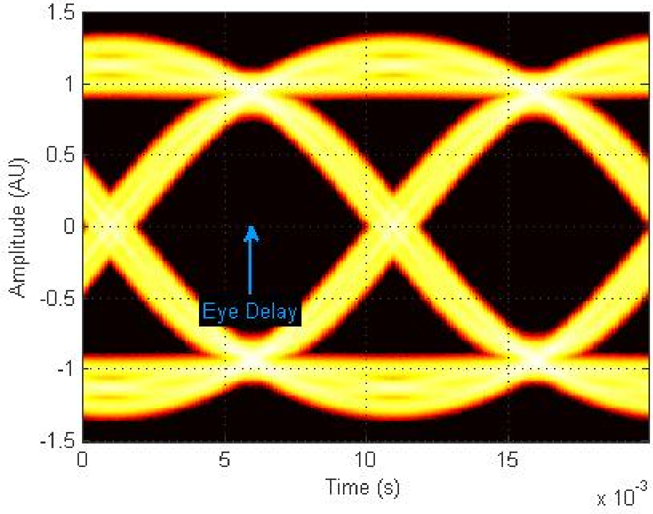


Note When an eye crossing time measurement falls within the [-0.5/Fs, 0) seconds interval, the time measurement wraps to the end of the eye diagram, i.e., the measurement wraps by 2*Ts seconds (where Ts is the symbol time). For a complex signal case, the analyze method issues a warning if the crossing time measurement of the in-phase branch wraps while that of the quadrature branch does not (or vice versa).

To avoid the time-wrapping or a warning, add a half-symbol duration delay to the current value in the MeasurementDelay property of the eye diagram object. This additional delay repositions the eye in the approximate center of the scope.

Eye Delay (EyeDelay)

Eye Delay is the distance from the midpoint of the eye to the time origin, measured in seconds. The analyze method calculates this distance using the crossing time. For a symmetric signal, EyeDelay is also the best sampling point.

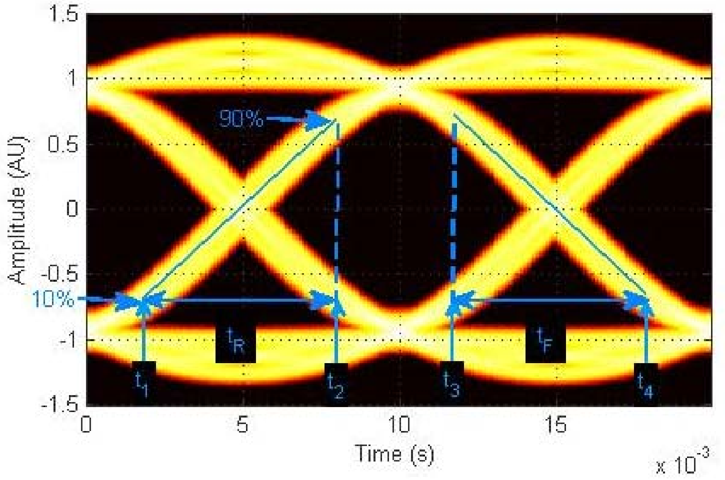


Eye Fall Time (EyeFallTime)

Eye Fall Time is the mean time between the high and low threshold values defined by the AmplitudeThreshold property of the eyemeasurementsetup object. The previous figure shows the fall time calculated from 10% to 90% of the eye amplitude.

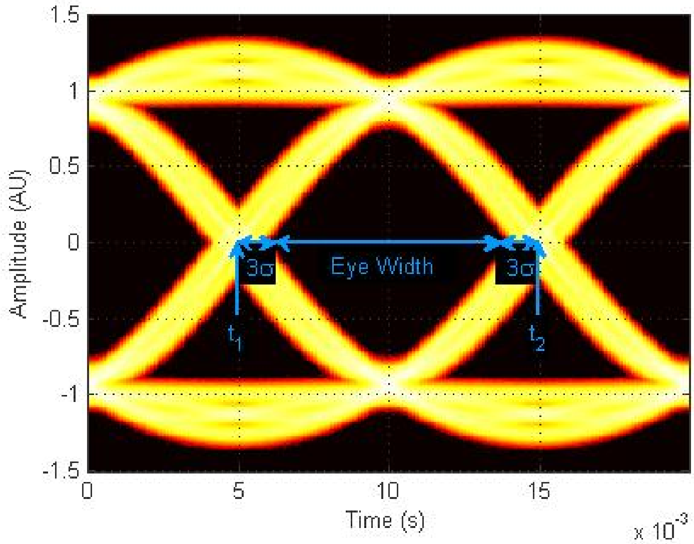
Eye Rise Time (EyeRiseTime)

Eye Rise Time is the mean time between the low and high threshold values defined by the AmplitudeThreshold property of the eyemeasurementsetup object. The following figure shows the rise time calculated from 10% to 90% of the eye amplitude.



Eye Width (EyeWidth)

Eye Width is the horizontal distance between two points that are three standard deviations (3σ) from the mean eye crossing times, towards the center of the eye. The value for *Eye Width* measurements is seconds.



Horizontal Eye Opening (EyeOpeningHor)

Horizontal Eye Opening is the horizontal distance between two points on the horizontal histogram that correspond to the *BER* value defined by the *BERThreshold* property of the eyemeasurementsetup object. The measurement is take at the amplitude value defined by the ReferenceAmplitude property of the eyemeasurementsetup object. It is calculated taking into account the random and deterministic components using a dual-Dirac model [5] (see the Dual Dirac Section).

A typical *BER* value for the eye opening measurements is 10^{-12}, which approximately corresponds to the 7σ point assuming a Gaussian distribution.

Peak-to-Peak Jitter (JitterP2P)

Peak-To-Peak Jitter is the difference between the extreme data points of the histogram.

Random Jitter (JitterRandom)

Random Jitter is defined as the Gaussian unbounded component of the jitter. The analyze method calculates it using the tail standard deviation estimated using the dual-Dirac method as follows [5]:

$$RJ = (Q_L + Q_R) * \sigma$$

where

$$Q_L = \sqrt{2} * erfc^{-1}\left(\frac{2 * BER}{\rho_L}\right)$$

and

$$Q_R = \sqrt{2} * erfc^{-1}\left(\frac{2 * BER}{\rho_R}\right)$$

BER is the bit error ratio at which the random jitter is calculated. It is defined with the *BERThreshold* property of the eyemeasuremensetup object.

RMS Jitter (JitterRMS)

RMS Jitter is the standard deviation of the jitter calculated from the horizontal histogram.

Total Jitter (JitterTotal)

Total Jitter is the sum of the random jitter and the deterministic jitter [5].

Measurement Setup Parameters

A number of set-up parameters control eye diagram measurements. This section describes these set-up parameters and the measurements they affect.

Eye Level Boundaries

Eye Level Boundaries are defined as a percentage of the symbol duration. The analyze method calculates the eye levels by averaging the vertical histogram within a given time interval defined by the eye level boundaries. A common value you can use for NRZ signals is 40% to 60%. For RZ signals, a narrower band of 5% is more appropriate. The default setting for *Eye level Boundaries* is a 2-by-1 vector where the first element is the lower boundary and the second element is the upper boundary. When the eye level boundary changes, the object recalculates this value.

Reference Amplitude

Reference Amplitude is the boundary value at which point the signal crosses from one signal level to another. Reference amplitude represents the decision boundary of the modulation scheme. This value is used to perform jitter measurements. The default setting for *Reference Amplitude* is a 2-by-1 double vector where the first element is the lower boundary and the second element is the upper boundary. Setting the reference amplitude resets the eye diagram.

The crossing instants of the input signal are detected and recorded as crossing times. A common value you can use for NRZ signals is `0`. For RZ signals, you can use the mean value of `1` and `0` levels. Reference amplitude is stored in a 2-by-N matrix, where the first row is the in-phase values and second row is the quadrature values. See Eye Crossing Time for more information.

Crossing Bandwidth

Crossing Bandwidth is the amplitude band used to measure the crossing times of the eye diagram. *Crossing Bandwidth* represents a percentage of the amplitude span of the eye diagram, typically 5%. See Eye Crossing Time for more information. The default setting for *Crossing Bandwidth* is 0.0500.

Bit Error Rate Threshold

The eye opening measurements, random, and total jitter measurements are performed at a given BER value. This BER value defines the BER threshold. A typical value is $1e^{-12}$. The default setting for *Bit Error Threshold* is $1.0000e^{-12}$. When the bit error rate threshold changes, the object recalculates this value.

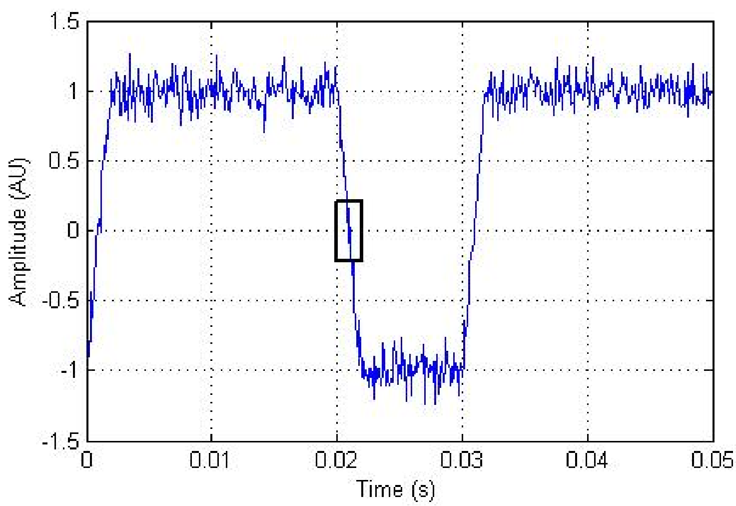
Amplitude Threshold

The rise time of the signal is defined as the time required for the signal to travel from the lower amplitude threshold to the upper amplitude threshold. The fall time, measured from the upper amplitude threshold to the lower amplitude threshold, is defined as a percentage of the eye amplitude. The default setting is 10% for the lower threshold and 90% for the upper threshold. Setting the amplitude threshold resets the eye diagram. See Eye Rise Time and Eye Fall Time for more information.

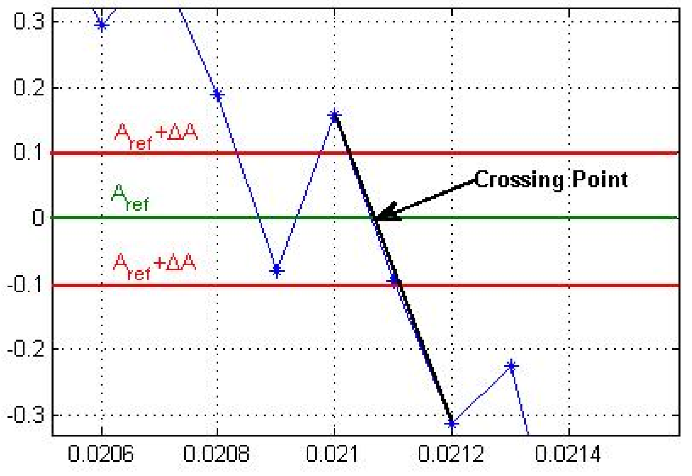
Jitter Hysteresis

You can use the *JitterHysteresis* property of the `eyemeasurementsetup` object to remove the effect of noise from the horizontal histogram estimation. The default value for *Jitter Hysteresis* is zero. Setting the jitter hysteresis value resets the eye diagram.

If channel noise impairs the signal being tested, as shown in the following figure, the signal may seem like it crosses the reference amplitude level multiple times during a single 0-1 or 1-0 transition.



See the zoomed—in image for more detail.



To eliminate the effect of noise, define a hysteresis region between two threshold values: $A_{ref} + \Delta A$ and $A_{ref} - \Delta A$, where A_{ref} is the reference amplitude value and ΔA is the jitter hysteresis value. If the signal crosses both threshold values, level crossing is declared. Then, linear interpolation calculates the crossing point in the horizontal histogram estimation.

Examples

```
% Construct an eye diagram object for signals in the range
% of [-3 3]
h = commscope.eyediagram('MinimumAmplitude', -3, ...
    'MaximumAmplitude', 3)

% Construct an eye diagram object for a signal with
% 1e-3 seconds of transient time
h = commscope.eyediagram('MeasurementDelay', 1e-3)

% Construct an eye diagram object for '2D Line' plot type
% with 100 traces to display
h = commscope.eyediagram('PlotType', '2D Line', ...
                         'NumberOfStoredTraces', 100)
```

References

[1] Nelson Ou, et al, *Models for the Design and Test of Gbps-Speed Serial Interconnects,*IEEE Design & Test of Computers, pp. 302-313, July-August 2004.

[2] HP E4543A Q Factor and Eye Contours Application Software, Operating Manual, http://agilent.com

[3] Agilent 71501D Eye-Diagram Analysis, User's Guide, http://www.agilent.com

[4] 4] Guy Foster, *Measurement Brief: Examining Sampling Scope Jitter Histograms,* White Paper, SyntheSys Research, Inc., July 2005.

[5] *Jitter Analysis: The dual-Dirac Model, RJ/DJ, and Q-Scale,* White Paper, Agilent Technologies, December 2004, http://www.agilent.com

See Also

`commscope`

Purpose

Create Scatter Plot scope

Syntax

```
h = commscope.ScatterPlot
h = commscope.ScatterPlot('PropertyName',PropertyValue,...)
```

Description

`commscope.ScatterPlot` collects data and displays results in a Figure window. You can create a scatter plot using a default configuration or by defining properties.

`h = commscope.ScatterPlot` returns a scatter plot scope, *h*.

`h = commscope.ScatterPlot('PropertyName',PropertyValue,...)` returns a scatter plot scope, *h*, with property values set to `PropertyValues`. See the Properties section of this help page for valid PropertyNames.

Properties

A ScatterPlot object has the properties shown on the following table. All properties are writable except for the ones explicitly noted otherwise.

Property	Description
Type	'Scatter Plot'. This is a read-only property.
SamplingFrequency	Sampling frequency of the input signal in Hz.
SamplesPerSymbol	Number of samples used to represent a symbol.
SymbolRate	The symbol rate of the input signal. This property is read-only and is automatically computed based on `SamplingFrequency` and `SamplesPerSymbol`.
MeasurementDelay	The time in seconds the scope will wait before starting to collect data.

Property	Description
SamplingOffset	The number of samples skipped at each sampling point relative to the `MeasurementDelay`.
Constellation	Expected constellation of the input signal.
RefreshPlot	The switch that controls the plot refresh style. The choices are: • ’on’ - The scatter plot refreshes every time the update method is called. • ’off’ - The scatter plot does not refresh when the update method is called.
SamplesProcessed	The number of samples processed by the scope. This value does not include the discarded samples during the `MeasurementDelay` period. This property is read-only.
PlotSettings	Plot settings control the scatter plot figure. • SymbolStyle - Line style of symbols • SignalTrajectory - The switch to control the visibility of the signal trajectory. The choices are ’on’ or ’off”. • SignalTrajectoryStyle - Line style of signal trajectory • Constellation - The switch to control the visibility of

Property	Description
	the constellation points. The choices are ’on’ or ’off”. • ConstellationStyle - Line style of signal trajectory • Grid - The switch to control the visibility of the grid. The choices are ’on’ or ’off”.

Methods

A Scatter Plot has the following methods.

autoscale

This method automatically scales the plot figure so its entire contents displays.

close

This method closes the scatter plot figure.

disp

This method displays the scatter plot properties.

plot

This method creates a scatter plot figure. If a figure exists, this method updates the figure’s contents.

`plot(h)` plots a scatter plot figure using default settings.

reset

This method resets the collected data of the scatter plot object.

`reset(h)` resets the collected data of the scatter plot object `h`. Resetting `h` also clears the plot and `NumberOfSymbols`.

update

This method updates the collected data of the scatter plot.

`update(h, r)` updates the collected data of the scatter plot, where *h* is the handle of the scatter plot object and *r* is the complex input data under test. This method updates the collected data and the plot (if `RefreshPlot` is true).

How To

- Viewing Signals Using Scatter Plots

Purpose

Construct pattern generator object

Syntax

```
h = commsrc.pattern
```

Description

`h = commsrc.pattern` constructs a pattern generator object, h. This syntax is equivalent to:

```
h = commsrc.pattern('SamplingFrequency', 10000, ...
                    'SamplesPerSymbol', 100, ...
                    'PulseType', 'NRZ', ...
                    'OutputLevels', [-1 1], ...
                    'RiseTime', 0, ...
                    'FallTime', 0, ...
                    'DataPattern', 'PRBS7', ...
                    'Jitter', commsrc.combinedjitter)
```

The pattern generator object produces modulated data patterns. This object can also inject jitter into the modulated signal.

Properties

A pattern generator object has the properties shown on the following table. You can edit all properties, except those explicitly noted otherwise.

Property	Description
`Type`	Type of pattern generator object (`'Pattern Generator'`). This property is not writable.
`SamplingFrequency`	Sampling frequency of the input signal in hertz.
`SymbolRate`	The symbol rate of the input signal. This property depends upon the `SamplingFequency` and `SamplesPerSymbol` properties. This property is not writable.

Property	Description
SamplesPerSymbol	The number of samples representing a symbol. SamplesPerSymbol must be an integer. This property affects SymbolRate.
PulseType	The type of pulse the object generates. Pulse types available: return-to-zero (RZ) and nonreturn-to-zero (NRZ). The initial condition for an NRZ pulse is 0.
OutputLevels	Amplitude levels that correspond to the symbol indices. For an NRZ pulse, this is a 1-by-2 vector. The first element of the 1-by-2 vector corresponds to the 0th symbol (data bit value 0). The second element corresponds to the 1st symbol (data bit value 1). For an RZ pulse, this is a scalar and the value corresponds to the data bit value 1.
DutyCycle	The duty cycle of the pulse the object generates. Displays calculated duty cycle based on pulse parameters. This property is not writable.
RiseTime	Specifies 10% to 90% rise time of the pulse in seconds.
PulseDuration	Pulse duration in seconds defined by IEEE STD 181 standard. (See the Return-to-Zero (RZ) Signal Conversion: Ideal Pulse to STD–181 figure in the Methods section.) Setting PulseType to return-to-zero enables this property.
FallTime	Specifies 10% to 90% fall time of the pulse in seconds.

Property	Description
`DataPattern`	The bit sequence the object uses. The following patterns are available: PRBS5 to PRBS15, PRBS23, PRBS31, and User Defined.
`UserDataPattern`	User-defined bit pattern consisting of a vector of ones and zeroes. Setting data pattern to `user defined` enables this property.
`Jitter`	Specifies jitter characteristics. Use this property to configure Random, Periodic and Dual Dirac Jitter.

Methods

A pattern generator object has five methods, as described in this section.

generate

This method outputs a frame worth of modulated and interpolated symbols. It has one input argument, which is the number of symbols in a frame. Its output is a double-column vector. You can call this method using the following syntax

```
x = generate(h, N)
```

where *h* is the handle to the object, *N* is the number of output symbols, and *x* is a double-column vector.

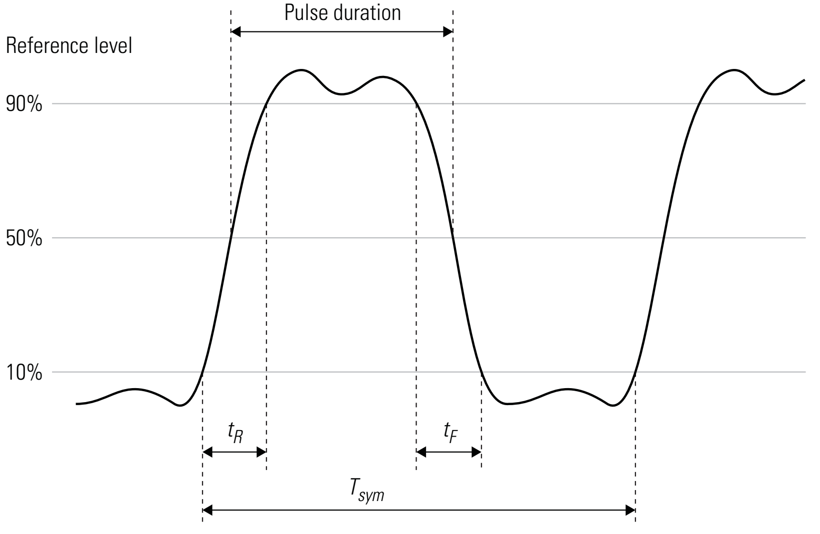
reset

This method resets the pattern generator to its default state. The property values do not reset unless they relate to the state of the object. This method has no input arguments.

idealtostd181

This method converts the ideal pulse specifications to IEEE STD-181 specifications: 0% to 100% rise time (TR) and fall time (TF) convert to 10% to 90% rise and fall times with a 50% pulse width duration, as shown in the following figure. This method also sets the appropriate properties.

idealtostd181(t_R, t_F, PW)



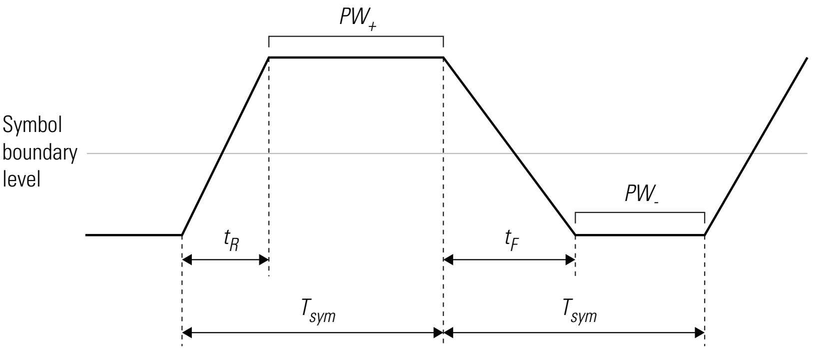
IEEE STD-181 Return-to-Zero (RZ) Signal Parameters

std181toideal

This method converts the IEEE STD-181 pulse specifications, stored in the pattern generator, to ideal pulse specifications. This method converts the 10% to 90% rise and fall times to 0% to 100% rise and fall times (*TR* and *TF*). It also converts the 50% pulse duration to pulse width (as shown in the following figure). Use the property values for IEEE STD-181 specifications

```
[tr tf pw] = stdstd181toideal(h)
```

where h is the pattern generator object handle and t_R is 0 to 100% rise time.



Ideal Pulse Non-Return-to-Zero (NRZ) Signal Parameters

computedcd

Computes the duty cycle distortion, DCD, of the pulse defined by the pattern generator object *h*.

DCD represents the ratio of the pulse on duration to the pulse off duration. For an NRZ pulse, on duration is the duration the pulse spends above the symbol boundary level. Off duration is the duration the pulse spends below zero.

```
dcd = computedcd(h)
```

The software calculates DCD given t_R, t_F, T_{sym}. This formula assumes that the symbol boundary level is zero.

$$T_h = (A_h\text{-}A_l) * \frac{t_R}{A_l} + (A_h\text{-}A_l) * \frac{t_F}{A_l} + \mathrm{PW}_+$$

$$T_l = (A_h\text{-}A_l) * \frac{t_R}{A_l} + (A_h\text{-}A_l) * \frac{t_F}{A_l} + \mathrm{PW}_-$$

$$\mathrm{DCD} = \frac{T_h}{T_l}$$

Where T_h is the duration of the high signal, T_l is the duration of the low signal, and DCD represents the ratio of the duration of the high signal to the low signal.

Purpose

Create PN sequence generator package

Syntax

```
h = commsrc.pn
h = commsrc.pn(property1,value1,...)
```

Description

`h = commsrc.pn` creates a default PN sequence generator object *h*, and is equivalent to the following:

```
H = COMMSRC.PN('GenPoly',       [1 0 0 0 0 1 1], ...
              'InitialStates', [0 0 0 0 0 1],   ...
              'CurrentStates', [0 0 0 0 0 1],   ...
              'Mask',          [0 0 0 0 0 1],   ...
              'NumBitsOut',    1)
```

or

```
H = COMMSRC.PN('GenPoly',       [1 0 0 0 0 1 1], ...
              'InitialStates', [0 0 0 0 0 1],   ...
              'CurrentStates', [0 0 0 0 0 1],   ...
              'Shift',         0,               ...
              'NumBitsOut',    1)
```

`h = commsrc.pn(property1,value1,...)` creates a PN sequence generator object, *h*, with properties you specify as property/value pairs.

Properties

A PN sequence generator has the properties shown on the following table. All properties are writable except for the ones explicitly noted otherwise.

Property	Description
GenPoly	Generator polynomial vector array of bits; must be descending order
InitialStates	Vector array (with length of the generator polynomial order) of initial shift register values (in bits)
CurrentStates	Vector array (with length of the generator polynomial order) of present shift register values (in bits)
NumBitsOut	Number of bits to output at each `generate` method invocation
`Mask` or `Shift`	A mask vector of binary 0 and 1 values is used to specify which shift register state bits are XORed to produce the resulting output bit value. Alternatively, a scalar shift value may be used to specify an equivalent shift (either a delay or advance) in the output sequence.

The `'GenPoly'` property values specify the shift register connections. Enter these values as either a binary vector or a vector of exponents of the nonzero terms of the generator polynomial in descending order of powers. For the binary vector representation, the first and last elements of the vector must be 1. For the descending-ordered polynomial representation, the last element of the vector must be 0. For more information and examples, see the LFSR SSRG Details section of this page.

Methods

A PN sequence generator is equipped with the following methods.

generate

Generate [NumBitsOut x 1] PN sequence generator values

reset

Set the `CurrentStates` values to the `InitialStates` values

getshift

Get the actual or equivalent `Shift` property value

getmask

Get the actual or equivalent `Mask` property value

copy

Make an independent copy of a `commsrc.pn` object

disp

Display PN sequence generator object properties

Side Effects of Setting Certain Properties

Setting the GenPoly Property

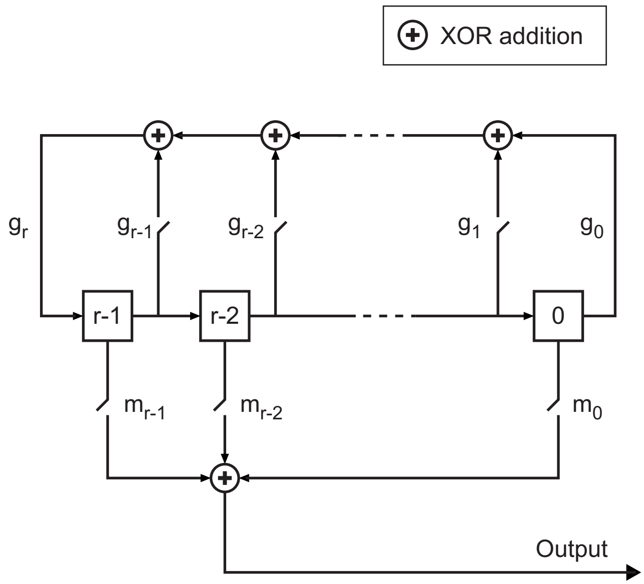
Every time this property is set, it will reset the entire object. In addition to changing the polynomial values, `'CurrentStates'`, `'InitialStates'`, and `'Mask'` will be set to their default values (`'NumBitsOut'` will remain the same), and no warnings will be issued.

Setting the InitialStates Property

Every time this property is set, it will also set `'CurrentStates'` to the new `'InitialStates'` setting.

LFSR SSRG Details

The `generate` method produces a pseudorandom noise (PN) sequence using a linear feedback shift register (LFSR). The LFSR is implemented using a simple shift register generator (SSRG, or Fibonacci) configuration, as shown below.



All *r* registers in the generator update their values at each time step according to the value of the incoming arrow to the shift register. The adders perform addition modulo 2. The shift register is described by the `'GenPoly'` property (generator polynomial), which is a primitive binary polynomial in *z*, $g_r z^r + g_{r-1} z^{r-1} + g_{r-2} z^{r-2} + ... + g_0$. The coefficient g_k is 1 if there is a connection from the kth register, as labeled in the preceding diagram, to the adder. The leading term g_r and the constant term g_0 of the `'GenPoly'` property must be 1 because the polynomial must be primitive.

You can specify the **Generator polynomial** parameter using either of these formats:

- A vector that lists the coefficients of the polynomial in descending order of powers. The first and last entries must be 1. Note that the length of this vector is one more than the degree of the generator polynomial.
- A vector containing the exponents of *z* for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`.

For example, `[1 0 0 0 0 0 1 0 1]` and `[8 2 0]` represent the same polynomial, p(z) = $z^8 + z^2 + 1$.

The **Initial states** parameter is a vector specifying the initial values of the registers. The **Initial states** parameter must satisfy these criteria:

- All elements of the **Initial states** vector must be binary numbers.
- The length of the **Initial states** vector must equal the degree of the generator polynomial.

Note At least one element of the **Initial states** vector must be nonzero in order for the block to generate a nonzero sequence. That is, the initial state of at least one of the registers must be nonzero.

For example, the following table indicates two sets of parameter values that correspond to a generator polynomial of p(z) = $z^8 + z^2 + 1$.

Quantity	Example 1	Example 2
Generator polynomial	`g1 = [1 0 0 0 0 0 1 0 1]`	`g2 = [8 2 0]`
Degree of generator polynomial	8, which is `length(g1)-1`	8
Initial states	`[1 0 0 0 0 0 1 0]`	`[1 0 0 0 0 0 1 0]`

Output mask vector (or scalar shift value) shifts the starting point of the output sequence. With the default setting for this parameter, the only connection is along the arrow labeled m_0, which corresponds to a shift of 0. The parameter is described in greater detail below.

You can shift the starting point of the PN sequence with **Output mask vector (or scalar shift value)**. You can specify the parameter in either of two ways:

- An integer representing the length of the shift
- A binary vector, called the *mask vector*, whose length is equal to the degree of the generator polynomial

The difference between the block's output when you set **Output mask vector (or scalar shift value)** to 0, versus a positive integer d, is shown in the following table.

	T = 0	T = 1	T = 2	...	T = d	T = d+1
Shift = 0	x_0	x_1	x_2	...	x_d	x_{d+1}
Shift = d	x_d	x_{d+1}	x_{d+2}	...	x_{2d}	x_{2d+1}

Alternatively, you can set **Output mask vector (or scalar shift value)** to a binary vector, corresponding to a polynomial in *z*, $m_{r-1}z^{r-1} + m_{r-2}z^{r-2} + \ldots + m_1z + m_0$, of degree at most r-1. The mask vector corresponding to a shift of d is the vector that represents $m(z) = z^d$ modulo g(*z*), where g(*z*) is the generator polynomial. For example, if the degree of the generator polynomial is 4, then the mask vector corresponding to d = 2 is `[0 1 0 0]`, which represents the polynomial $m(z) = z^2$. The preceding schematic diagram shows how **Output mask vector (or scalar shift value)** is implemented when you specify it as a mask vector. The default setting for **Output mask vector (or scalar shift value)** is `0`. You can calculate the mask vector using the Communications System Toolbox function `shift2mask`.

Sequences of Maximum Length

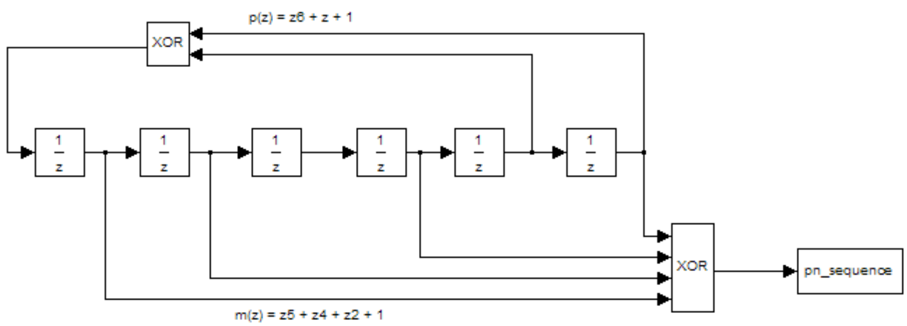
If you want to generate a sequence of the maximum possible length for a fixed degree, *r*, of the generator polynomial, you can set **Generator polynomial** to a value from the following table. See Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995 for more information about the shift-register configurations that these polynomials represent.

r	Generator Polynomial	r	Generator Polynomial
2	[2 1 0]	21	[21 19 0]
3	[3 2 0]	22	[22 21 0]
4	[4 3 0]	23	[23 18 0]
5	[5 3 0]	24	[24 23 22 17 0]
6	[6 5 0]	25	[25 22 0]
7	[7 6 0]	26	[26 25 24 20 0]
8	[8 6 5 4 0]	27	[27 26 25 22 0]
9	[9 5 0]	28	[28 25 0]
10	[10 7 0]	29	[29 27 0]
11	[11 9 0]	30	[30 29 28 7 0]
12	[12 11 8 6 0]	31	[31 28 0]
13	[13 12 10 9 0]	32	[32 31 30 10 0]
14	[14 13 8 4 0]	33	[33 20 0]
15	[15 14 0]	34	[34 15 14 1 0]
16	[16 15 13 4 0]	35	[35 2 0]
17	[17 14 0]	36	[36 11 0]
18	[18 11 0]	37	[37 12 10 2 0]
19	[19 18 17 14 0]	38	[38 6 5 1 0]
20	[20 17 0]	39	[39 8 0]
40	[40 5 4 3 0]	47	[47 14 0]
41	[41 3 0]	48	[48 28 27 1 0]
42	[42 23 22 1 0]	49	[49 9 0]
43	[43 6 4 3 0]	50	[50 4 3 2 0]
44	[44 6 5 2 0]	51	[51 6 3 1 0]

r	Generator Polynomial	r	Generator Polynomial
45	[45 4 3 1 0]	52	[52 3 0]
46	[46 21 10 1 0]	53	[53 6 2 1 0]

Examples

Setting up the PN sequence generator



This figure defines a PN sequence generator with a generator polynomial $p(z) = z^6 + z + 1$. You can set up the PN sequence generator by typing the following at the MATLAB command line:

```
h1 = commsrc.pn('GenPoly', [1 0 0 0 0 1 1], 'Mask', [1 1 0 1 0 1]);
h2 = commsrc.pn('GenPoly', [1 0 0 0 0 1 1], 'Shift', 22);
mask2shift ([1 0 0 0 0 1 1],[1 1 0 1 0 1])
```

The output of the example is given below:

```
ans =

    22
```

Alternatively, you can input GenPoly as the exponents of z for the nonzero terms of the polynomial in descending order of powers:

```
h = commsrc.pn('GenPoly', [6 1 0], 'Mask', [1 1 0 1 0 1])
```

General Use of commsrc.pn

The following is an example of typical usage:

```
% Construct a PN object
h = commsrc.pn('Shift', 0);

% Output 10 PN bits
set(h, 'NumBitsOut', 10);
generate(h)

% Output 10 more PN bits
generate(h)

% Reset (to the initial shift register state values)
reset(h);

% Output 4 PN bits
set(h, 'NumBitsOut', 4);
generate(h)
```

Behavior of a Copied commsrc.pn Object

When a commsrc.pn object is copied, its states are also copied. The subsequent outputs, therefore, from the copied object are likely to be different from the initial outputs from the original object. The following code illustrates this behavior:

```
h = commsrc.pn('Shift', 0);
set(h, 'NumBitsOut', 5);
generate(h)
```

`h` generates the sequence:

```
     1
     0
     0
     0
     0
```

However, if h is copied to g, and g is made to generate a sequence:

```
g=copy(h);
generate(g)
```

the generated sequence is different from that initially generated from h:

```
     0
     1
     0
     0
     0
```

This difference occurs because the state of h having generated 5 bits was copied to g. If g is reset:

```
reset(g);
generate(g)
```

then it generates the same sequence that h did:

```
     1
     0
     0
     0
     0
```

See Also

mask2shift | shift2mask

Purpose

Default Simulink model settings for Communications System Toolbox software

Syntax

`commstartup`

Description

`commstartup` changes the default Simulink model settings to values more appropriate for the simulation of communication systems. The changes apply to new models that you create later in the MATLAB® session, but not to previously created models.

Note The DSP System Toolbox™ application includes a similar dspstartup script, which assigns different model settings. For modeling communication systems, you should use `commstartup` alone.

To install the communications-related model settings each time you start MATLAB, invoke `commstartup` from your `startup.m` file.

To be more specific, the settings in `commstartup` cause models to:

- Use the variable-step discrete solver in single-tasking mode
- Use starting and ending times of `0` and `Inf`, respectively
- Avoid producing a warning or error message for inherited sample times in source blocks
- Set the Simulink Boolean logic signals parameter to `Off`
- Avoid saving output or time information to the workspace
- Produce an error upon detecting an algebraic loop
- Inline parameters if you use the Model Reference feature of Simulink

If your communications model does not work well with these default settings, you can change each of the individual settings as the model requires.

See Also

`startup`

commtest.ErrorRate

Purpose

Create error rate test console

Syntax

```
h = commtest.ErrorRate
h = commtest.ErrorRate(sys)
h = commtest.ErrorRate(sys,'PropertyName',PropertyValue,...)
h = commtest.ErrorRate('PropertyName',PropertyValue,...)
```

Description

`h = commtest.ErrorRate` returns an error rate test console, h. The error rate test console runs simulations of a system under test to obtain error rates.

`h = commtest.ErrorRate(sys)` returns an error rate test console, error rate test console, h, with each specified property set to the h, with an attached system under test, SYS.

`h = commtest.ErrorRate(sys,'PropertyName',PropertyValue,...)` returns an error rate test console, h, with an attached system under test, sys. Each specified property, 'PropertyName', is set to the specified value, PropertyValue.

`h = commtest.ErrorRate('PropertyName',PropertyValue,...)` returns an error rate test console, h, with each specified property 'PropertyName', set to the specified value, PropertyValue.

Properties

The error rate test console object has the properties in the following table. Setting any property resets the object. A property that is *irrelevant* is one that you can set, but its value does not affect measurements. Similarly, you cannot display irrelevant properties using the `disp` method. You can write to all properties, except for the ones explicitly noted otherwise.

Property	Description
Description	'Error Rate Test Console'. Read-only.
SystemUnderTestName	System under test name. Read-only.

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>FrameLength</td><td>Specify the length of the transmission frame at each iteration. This property becomes relevant only when the system under test registers a valid test input.

If the system under test registers a <code>NumTransmissions</code> test input and calls its <code>getInput</code> method, the error rate test console returns the value stored in <code>FrameLength</code>. Using an internal data source, the system under test uses this value to generate a transmission frame of the specified length.
If the system under test registers a <code>DiscreteRandomSource</code> test input and calls its <code>getInput</code> method, the test console generates and returns a frame of symbols. The length of the frame of symbols matches the <code>FrameLength</code> property. This property defaults to 500.
</td></tr>
<tr><td>IterationMode</td><td>Specify how the object determines simulation points.

If set to <code>Combinatorial</code>, the object performs simulations for all possible combinations of registered test parameter sweep values.
If set to <code>Indexed</code>, the object performs simulations for all indexed sweep value sets. The i^{th} sweep value set consists of the i^{th} element of every sweep value vector for each registered test parameter. All sweep value vectors must have equal length, except for values that are unit length.

Note that for the following sweep parameter settings:

Parameter1 = [a_1 a_2]
Parameter2 = [b_1 b_2]
Parameter3 = [c_1]

In Indexed Mode, the test console performs simulations for the following sweep parameter sets:</td></tr>
</table>

Property	Description
	(a_1, b_1, c_1) (a_2, b_2, c_1) In Combinatorial Mode, the test console performs simulations for the following sweep parameter sets: (a_1, b_1, c_1) (a_1, b_2, c_1) (a_2, b_1, c_1) (a_2, b_2, c_1)
SystemResetMode	Specify the stage of a simulation run at which the system resets. • Setting to `Reset at new simulation point` resets the system under test at the beginning of a new simulation point. • Setting to `Reset at every iteration` resets the system under test at every iteration.

SimulationLimitOption	Specify how to stop the simulation for each sweep parameter point.
	• If set to `Number of transmissions` the simulation for a sweep parameter point stops when the number of transmissions equals the value for `MaxNumTransmissions`.
	  - Set `TransmissionCountTestPoint` to the name of the registered test point containing the transmission count you are comparing to `MaxNumTransmissions`.
	• If set to `Number of errors` the simulation for a sweep parameter point stops when the number of errors equals the value for `MinNumErrors`.
	  - Set the `ErrorCountTestPoint` to the name of the registered test point containing the error count you are comparing to the `MinNumErrors`.

Property	Description
	• Setting to `Number of errors or transmissions` stops the simulation for a sweep parameter point when meeting one of two conditions. - The simulation stops when the number of transmissions equals the value for `MaxNumTransmissions`. - The simulation stops when obtaining the number of errors matching `NumErrors`. • Setting this property to `Number of errors and transmissions` stops the simulation for a sweep parameter point when meeting the following condition. - The simulation stops when the number of transmissions *and* the number errors have at least reached the values in `MinNumTransmissions` and `MinNumErrors`. Set `TransmissionCountTestPoint` to the name of the registered test point that contains the transmission count you are comparing to the `MaxNumTransmissions` property. To control the simulation length, set `ErrorCountTestPoint` to the name of the registered test point containing the error count you are comparing to `MinNumErrors`. Call the `info` method of the error rate test console to see the valid registered test point names.

Property	Description

MaxNumTransmissions	Specify the maximum number of transmissions the object counts before stopping the simulation for a sweep parameter point. This property becomes relevant only when `SimulationLimitOption` is `Number of transmissions` or `Number of errors or transmissions`.
	
	• When setting `SimulationLimitOption` to `Number of transmissions` the simulation for each sweep parameter point stops when reaching the number of transmissions `MaxNumTransmissions` specifies.
	• Setting `SimulationLimitOption` to `Number of errors or transmissions` stops the simulation for each sweep parameter point for one of two conditions.
	  - The simulation stops when completing the number of transmissions `MaxNumTransmissions` specifies.
	  - The simulation stops when obtaining the number of errors `MinNumErrors` specifies.
	
	The `TransmissionCountTestPoint` property supplies the name of a registered test point containing the count transmission type. Calling the `info` method of the error rate test console displays the valid registered test points. If this property contains registered test points, the test console runs iterations equal to the value for `MaxNumTransmissions` for each sweep parameter point. If this property has no registered test parameters, the test console runs the number of iterations equal to the value for `MaxNumTransmissions` and stops. The value defaults to 1000.

Property	Description

MinNumErrors	Specify the minimum number of errors the object counts before stopping the simulation for a sweep parameter point. This property becomes relevant only when setting the `SimulationLimitOption` to `Number of errors` or `Number of errors or transmissions`.
	
	• When setting `SimulationLimitOption` to `Number of errors` the simulation for each parameter point stops when reaching the number of errors you specify for the `MinNumErrors` property.
	• When setting the `SimulationLimitOption` property to `Number of errors or transmissions` the simulation for each sweep parameter point stops for one of two conditions.
	  - The simulation stops when reaching the number of errors you specify for the `MaxNumTransmissions` property.
	  - The simulation stops when reaching the number of errors you specify for the `MinNumErrors` property.
	
	Specify the type of errors the error count uses by setting the `ErrorCountTestPoint` property to the name of a registered test point containing the count. Call the `info` method of the error rate test console to see the valid registered test point names. This value defaults to 100.

Property	Description
TransmissionCountTestPoint	Specify and register a test point containing the transmission count that controls the test console simulation stop mechanism. This property becomes relevant only when setting `SimulationLimitOption` to `Number of transmissions`, `Number of errors or transmissions`, or `Number of errors and transmissions`. In this scenario, if you register a test point, and `TransmissionCountTestPoint` equals `Not set`, the value of this property automatically updates to that of the registered test point name. Call the `info` method to see the valid test point names.
ErrorCountTestPoint	Specify and register the name of a test point containing the error count that controls the simulation stop mechanism. This property is only relevant when setting the `SimulationLimitOption` property to `Number of errors`, `Number of errors or transmissions`, or `Number of errors and transmissions`. In this scenario, if you register a test point, and `ErrorCountTestPoint` equals `Not set`, the value of this property automatically updates to that of the registered test point name. Call the `info` method to see the valid test point names.

Methods

The error rate test console object has the following methods:

run

Runs a simulation.

Runs the number of error rate simulations you specify for a system under test with a specified set of parameter values. If a Parallel Computing Toolbox™ license is available and a matlabpool is open, then the object distributes the iterations among the number of workers available.

getResults

Returns the simulation results.

r = getResults(h) returns the simulation results, *r*, for the test console, *h*. *r* is an object of the type you specify using testconsole.Results. It contains the simulation data for all the registered test points and methods to parse the data and plot it.

info

Returns a report of the current test console settings.

info(h) displays the current test console settings, such as registered test parameters and registered test points.

reset

Resets the error rate test console.

reset(h) resets test parameters and test probes and then clears all simulation results of test console, *h*.

attachSystem

Attaches a system to test console.

attachSystem(ho,sys) attaches a valid user-defined system, sys, to the test console, h.

detachSystem

Detaches the system from the test console.

detachSystem(h) detaches a system from the test console, h. This method also clears the registered test inputs, test parameters, test probes, and test points.

setTestParameterSweepValues

Sets test parameter sweep values.

setTestParameterSweepValues(h,name,sweep) specifies a set of sweep values, 'sweep', for the registered test parameter, 'name', in the test console, h. You only specify sweep values for registered test parameters. sweep must have values within the specified range of the test parameter. It can be a row vector of numeric values, or a cell array of char values. Display the valid ranges using the getTestParameterValidRanges method.

setTestParameterSweepValues(h,name1,sweep1,name2,sweep2...) simultaneously specifies sweep values for multiple registered test parameters.

getTestParameterSweepValues

Returns test parameter sweep values.

getTestParameterSweepValues(h,name) gets the sweep values currently specified for the registered test parameter, name, in the test console, h.

getTestParameterValidRanges

Returns the test parameter valid ranges.

getTestParameterValidRanges(h,name) gets the valid ranges for a registered test parameter, name, in the test console, h.

registerTestPoint

Registers a test point.

registerTestPoint(h, name, actprobe,expprobe) registers a new test point object, name, to the error rate test console, h. The test point must contain a pair of registered test probes, actprobe, and expprobe. actprobe contains actual data, and expprobe contains expected data. The object compares the data from these probes and obtains error rate values. The error rate calculation uses a default error rate calculator function that simply performs one-to-one comparisons of the data vectors available in the probes.

registerTestPoint(h, name, actprobe,expprobe, handle) adds the handle, handle, to a user-defined error calculation function that compares the data in the probes and then obtains error rate results.

The user-defined error calculation function must comply with the following syntax: [ecnt tcnt] = functionName(act, exp, udata) where

- ecnt output corresponds to the error count

- tcnt output is the number of transmissions used to obtain the error count
- act and exp correspond to actual and expected data

The error rate test console sets the inputs to the data available in the pair of test point probes, actprobe, and expprobe.

udata is a data input that the system under test passes to the test console at run time, using the setUserData method. udata contains the data necessary to compute errors, such as delays and data buffers.

The error rate test console passes the data that the system under test logs to the error calculation functions for all the registered test points. Calling the info method returns the names of the registered test points and the error rate calculator functions associated with them. It also returns the names of the registered test probes.

unregisterTestPoint

Unregister a test point.

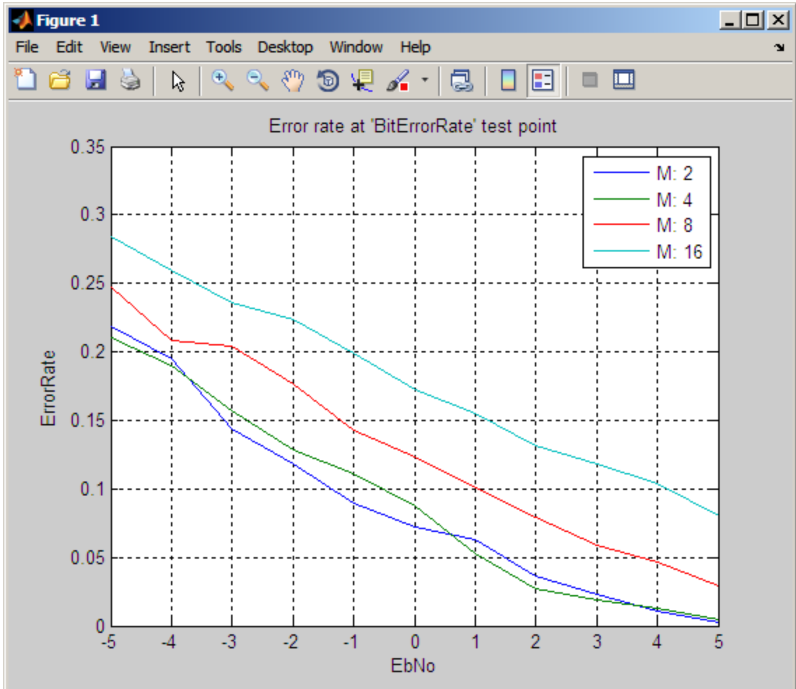
unregisterTestPoint(h,name) removes the test point, name, from the test console, h.

Examples

```
% Obtain bit error rate and  symbol error rate of an M-PSK system
% for different modulation orders and EbNo values.

% Instantiate an ErrorRate test console. The default error rate
% test console has an M-PSK system attached.
       h = commtest.ErrorRate;

% Set sweep values for simulation test parameters
setTestParameterSweepValues(h,'M',2.^[1 2 3 4])
setTestParameterSweepValues(h,'EbNo',(-5:5))

% Register test points
registerTestPoint(h,'SymbolErrorRate','TxInputSymbols',...,
'RxOutputSymbols')
registerTestPoint(h,'BitErrorRate','TxInputBits','RxOutputBits')
```

```
% Set simulation stop criteria.
h.TransmissionCountTestPoint = 'SymbolErrorRate';

% Get information about the simulation settings
info(h)

% Run the MPSK simulations
run(h)

% Get the results
R = getResults(h);

% Plot EbNo versus bit error rate for different values of modulation
% order M
R.TestParameter2 = 'M';
plot(R)
```

This example generates a figure similar to the following:



See Also

`testconsole.Results`

How To

- Running Simulations Using the Error Rate Test Console
- Error Rate Test Console

Purpose

Source code mu-law or A-law compressor or expander

Syntax

```
out = compand(in,param,v)
out = compand(in,Mu,v,'mu/compressor')
out = compand(in,Mu,v,'mu/expander')
out = compand(in,A,v,'A/compressor')
out = compand(in,A,v,'A/expander')
```

Description

`out = compand(in,param,v)` implements a μ-law compressor for the input vector `in`. `Mu` specifies μ, and `v` is the input signal's maximum magnitude. `out` has the same dimensions and maximum magnitude as `in`.

`out = compand(in,Mu,v,'mu/compressor')` is the same as the syntax above.

`out = compand(in,Mu,v,'mu/expander')` implements a μ-law expander for the input vector `in`. `Mu` specifies μ and `v` is the input signal's maximum magnitude. `out` has the same dimensions and maximum magnitude as `in`.

`out = compand(in,A,v,'A/compressor')` implements an A-law compressor for the input vector `in`. The scalar `A` is the A-law parameter, and `v` is the input signal's maximum magnitude. `out` is a vector of the same length and maximum magnitude as `in`.

`out = compand(in,A,v,'A/expander')` implements an A-law expander for the input vector `in`. The scalar `A` is the A-law parameter, and `v` is the input signal's maximum magnitude. `out` is a vector of the same length and maximum magnitude as `in`.

Note The prevailing parameters used in practice are $\mu = 255$ and $A = 87.6$.

Examples

The examples below illustrate the fact that compressors and expanders perform inverse operations.

```
compressed = compand(1:5,87.6,5,'a/compressor')
expanded = compand(compressed,87.6,5,'a/expander')
```

The output is

```
compressed =

    3.5296    4.1629    4.5333    4.7961    5.0000


expanded =

    1.0000    2.0000    3.0000    4.0000    5.0000
```

Algorithms

For a given signal x, the output of the μ-law compressor is

$$y = \frac{V\log(1+\mu|x|/V)}{\log(1+\mu)}\operatorname{sgn}(x)$$

where V is the maximum value of the signal x, μ is the μ-law parameter of the compander, log is the natural logarithm, and sgn is the signum function (`sign` in MATLAB).

The output of the A-law compressor is

$$y = \begin{cases} \dfrac{A|x|}{1+\log A}\operatorname{sgn}(x) & \text{for } 0 \le |x| \le \dfrac{V}{A} \\ \dfrac{V(1+\log(A|x|/V))}{1+\log A}\operatorname{sgn}(x) & \text{for } \dfrac{V}{A} < |x| \le V \end{cases}$$

where A is the A-law parameter of the compander and the other elements are as in the μ-law case.

References

[1] Sklar, Bernard, *Digital Communications: Fundamentals and Applications*, Englewood Cliffs, NJ, Prentice-Hall, 1988.

See Also

quantiz | dpcmenco | dpcmdeco

How To

- "Compand a Signal"

Purpose

Restore ordering of symbols using shift registers

Syntax

```
deintrlved = convdeintrlv(data,nrows,slope)
[deintrlved,state] = convdeintrlv(data,nrows,slope)
[deintrlved,state] = convdeintrlv(data,nrows,slope,
   init_state)
```

Description

`deintrlved = convdeintrlv(data,nrows,slope)` restores the ordering of elements in `data` by using a set of `nrows` internal shift registers. The delay value of the kth shift register is `(nrows-k)*slope`, where k = 1, 2, 3,..., `nrows`. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`[deintrlved,state] = convdeintrlv(data,nrows,slope)` returns a structure that holds the final state of the shift registers. `state.value` stores any unshifted symbols. `state.index` is the index of the next register to be shifted.

`[deintrlved,state] = convdeintrlv(data,nrows,slope,init_state)` initializes the shift registers with the symbols contained in `init_state.value` and directs the first input symbol to the shift register referenced by `init_state.index`. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding interleaver.

Using an Interleaver-Deinterleaver Pair

To use this function as an inverse of the `convintrlv` function, use the same `nrows` and `slope` inputs in both functions. In that case, the two functions are inverses in the sense that applying `convintrlv` followed by `convdeintrlv` leaves data unchanged, after you take their combined delay of `nrows*(nrows-1)*slope` into account. To learn more about delays of convolutional interleavers, see “Delays of Convolutional Interleavers”.

Examples

The example in "Effect of Delays on Recovery of Convolutionally Interleaved Data Using MATLAB" uses `convdeintrlv` and illustrates how you can handle the delay of the interleaver/deinterleaver pair when recovering data.

The example on the reference page for `muxdeintrlv` illustrates how to use the `state` output and `init_state` input with that function; the process is analogous for this function.

References

[1] Heegard, Chris, and Stephen B. Wicker, *Turbo Coding*, Boston, Kluwer Academic Publishers, 1999.

See Also

`convintrlv` | `muxdeintrlv`

How To

- "Interleaving"

Purpose

Convolutionally encode binary data

Syntax

```
code = convenc(msg,trellis)
code = convenc(msg,trellis,puncpat)
code = convenc(msg,trellis,...,init_state)
[code,final_state] = convenc(...)
```

Description

`code = convenc(msg,trellis)` encodes the binary vector `msg` using the convolutional encoder whose MATLAB trellis structure is `trellis`. For details about MATLAB trellis structures, see "Trellis Description of a Convolutional Code". Each symbol in `msg` consists of `log2(trellis.numInputSymbols)` bits. The vector `msg` contains one or more symbols. The output vector `code` contains one or more symbols, each of which consists of `log2(trellis.numOutputSymbols)` bits.

`code = convenc(msg,trellis,puncpat)` is the same as the syntax above, except that it specifies a puncture pattern, `puncpat`, to allow higher rate encoding. `puncpat` must be a vector of `1`s and `0`s, where the `0`s indicate the punctured bits. `puncpat` must have a length of at least `log2(trellis.numOutputSymbols)` bits.

`code = convenc(msg,trellis,...,init_state)` allows the encoder registers to start at a state specified by `init_state`. `init_state` is an integer between `0` and `trellis.numStates-1` and must be the last input parameter.

`[code,final_state] = convenc(...)` encodes the input message and also returns the encoder's state in `final_state`. `final_state` has the same format as `init_state`.

Examples

Encodes five two-bit symbols using a rate 2/3 convolutional code. A schematic of this encoder is on the `poly2trellis` reference page.

```
s = RandStream.create('mt19937ar', 'seed',123);
prevStream = RandStream.setGlobalStream(s); % Set stream for repeatabi
code1 = convenc(randi([0 1],10,1),...
poly2trellis([5 4],[23 35 0; 0 5 13]));
RandStream.setGlobalStream(prevStream); % Restore default stream
```

The following syntax defines the encoder's trellis structure explicitly and then uses `convenc` to encode 10 one-bit symbols. A schematic of this encoder is in "Trellis Description of a Convolutional Code".

```
trel = struct('numInputSymbols',2,'numOutputSymbols',4,...
'numStates',4,'nextStates',[0 2;0 2;1 3;1 3],...
'outputs',[0 3;1 2;3 0;2 1]);
code2 = convenc(randi([0 1],10,1),trel);
```

The following syntax illustrates how to use the final state and initial state arguments when invoking `convenc` repeatedly. Notice that `[code3; code4]` is the same as the earlier example's output, `code1`.

```
s = RandStream.create('mt19937ar', 'seed',123);
prevStream = RandStream.setGlobalStream(s); % Set stream for repeatabilit
trel = poly2trellis([5 4],[23 35 0; 0 5 13]);
msg = randi([0 1],10,1);
% Encode part of msg, recording final state for later use.
[code3,fstate] = convenc(msg(1:6),trel);
% Encode the rest of msg, using state as an input argument.
code4 = convenc(msg(7:10),trel,fstate);
RandStream.setGlobalStream(prevStream); % Restore default stream
```

Examples

For some commonly used puncture patterns for specific rates and polynomials, see the last three references.

References

[1] Clark, G. C. Jr. and J. Bibb Cain., *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

[2] Gitlin, Richard D., Jeremiah F. Hayes, and Stephen B. Weinstein, *Data Communications Principles*, New York, Plenum, 1992.

[3] Yasuda, Y., et. al., "High rate punctured convolutional codes for soft decision Viterbi decoding," *IEEE Transactions on Communications*, vol. COM-32, No. 3, pp 315–319, Mar. 1984.

[4] Haccoun, D., and G. Begin, “High-rate punctured convolutional codes for Viterbi and sequential decoding,” *IEEE Transactions on Communications*, vol. 37, No. 11, pp 1113–1125, Nov. 1989.

[5] Begin, G., et.al., “Further results on high-rate punctured convolutional codes for Viterbi and sequential decoding,” *IEEE Transactions on Communications*, vol. 38, No. 11, pp 1922–1928, Nov. 1990.

See Also

`distspec` | `vitdec` | `poly2trellis` | `istrellis`

How To

- “Convolutional Codes”

Purpose

Permute symbols using shift registers

Syntax

```
intrlved = convintrlv(data,nrows,slope)
[intrlved,state] = convintrlv(data,nrows,slope)
[intrlved,state] = convintrlv(data,nrows,slope,init_state)
```

Description

`intrlved = convintrlv(data,nrows,slope)` permutes the elements in `data` by using a set of `nrows` internal shift registers. The delay value of the kth shift register is `(k-1)*slope`, where k = 1, 2, 3,... `nrows`. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`[intrlved,state] = convintrlv(data,nrows,slope)` returns a structure that holds the final state of the shift registers. `state.value` stores any unshifted symbols. `state.index` is the index of the next register to be shifted.

`[intrlved,state] = convintrlv(data,nrows,slope,init_state)` initializes the shift registers with the symbols contained in `init_state.value` and directs the first input symbol to the shift register referenced by `init_state.index`. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding deinterleaver.

Examples

The example below shows that `convintrlv` is a special case of the more general function `muxintrlv`. Both functions yield the same numerical results.

```
x = randi([0 1],100,1); % Original data
nrows = 5; % Use 5 shift registers
slope = 3; % Delays are 0, 3, 6, 9, and 12.
y = convintrlv(x,nrows,slope); % Interleaving using convintrlv.
delay = [0:3:12]; % Another way to express set of delays
y1 = muxintrlv(x,delay); % Interleave using muxintrlv.
isequal(y,y1)
```

The output below shows that `y`, obtained using `convintrlv`, and `y1`, obtained using `muxintrlv`, are the same.

```
ans =

     1
```

Another example using this function is in “Effect of Delays on Recovery of Convolutionally Interleaved Data Using MATLAB”.

The example on the `muxdeintrlv` reference page illustrates how to use the `state` output and `init_state` input with that function; the process is analogous for this function.

References

[1] Heegard, Chris, and Stephen B. Wicker, *Turbo Coding*, Boston, Kluwer Academic Publishers, 1999.

See Also

`convdeintrlv` | `muxintrlv` | `helintrlv`

How To

- “Interleaving”

Purpose

Convolution matrix of Galois field vector

Syntax

```
A = convmtx(c,n)
```

Description

A *convolution matrix* is a matrix, formed from a vector, whose inner product with another vector is the convolution of the two vectors.

`A = convmtx(c,n)` returns a convolution matrix for the Galois vector `c`. The output `A` is a Galois array that represents convolution with `c` in the sense that `conv(c,x)` equals

- `A*x`, if `c` is a column vector and `x` is any Galois column vector of length `n`. In this case, `A` has `n` columns and `m+n-1` rows.
- `x*A`, if `c` is a row vector and `x` is any Galois row vector of length `n`. In this case, `A` has `n` rows and `m+n-1` columns.

Examples

The code below illustrates the equivalence between using the `conv` function and multiplying by the output of `convmtx`.

```
m = 4;
c = gf([1; 9; 3],m); % Column vector
n = 6;
x = gf(randi([0 2^m-1],n,1),m);
ck1 = isequal(conv(c,x), convmtx(c,n)*x) % True
ck2 = isequal(conv(c',x'),x'*convmtx(c',n)) % True
```

The output is

```
ck1 =

     1


ck2 =

     1
```

See Also

conv

How To

- "Signal Processing Operations in Galois Fields"

cosets

Purpose

Produce cyclotomic cosets for Galois field

Syntax

```
cst = cosets(m)
```

Description

`cst = cosets(m)` produces cyclotomic cosets mod `2^m-1`. Each element of the cell array `cst` is a Galois array that represents one cyclotomic coset.

A cyclotomic coset is a set of elements that share the same minimal polynomial. Together, the cyclotomic cosets mod `2^m-1` form a partition of the group of nonzero elements of GF(`2^m`). For more details on cyclotomic cosets, see the works listed in "References" on page 1-163.

Examples

The commands below find and display the cyclotomic cosets for GF(8). As an example of interpreting the results, `c{2}` indicates that A, A^2, and $A^2 + A$ share the same minimal polynomial, where A is a primitive element for GF(8).

```
c = cosets(3);
c{1}'
c{2}'
c{3}'
```

The output is below.

```
ans = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     1



ans = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     2     4     6
```

```
ans = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     3     5     7
```

References

[1] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, MA, Addison-Wesley, 1983, p. 105.

[2] Lin, Shu, and Daniel J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, Englewood Cliffs, NJ, Prentice-Hall, 1983.

See Also

minpol

Purpose

Construct CRC detector object

Syntax

```
h= crc.detector(polynomial)
```

```
h= crc.detector(generatorObj)
```

```
h= crc.detector(`Polynomial', polynomial, `param1', val1,
etc.)
```

```
h= crc.detector
```

Description

`h= crc.detector(polynomial)` constructs a CRC detector object H defined by the generator polynomial POLYNOMIAL

`h= crc.detector(generatorObj)` constructs a CRC detector object H defined by the parameters found in the CRC generator object GENERATOROBJ

`h= crc.detector(`property1', val1, ...)` constructs a CRC detector object H with properties as specified by PROPERTY/VALUE pairs.

`h= crc.detector` constructs a CRC detector object H with default properties. It constructs a CRC-CCITT detector, and is equivalent to:

```
h= crc.detector('Polynomial', '0x1021', 'InitialState',
'0xFFFF', 'ReflectInput', ...
```

```
false, 'ReflectRemainder', false, 'FinalXOR', '0x0000')
```

Properties

The following table describes the properties of a CRC detector object. All properties are writable, except `Type`.

Property	Description
`Type`	Specifies the object as a 'CRC Detector'.
`Polynomial`	The generator polynomial that defines connections for a linear feedback shift register. This property can be specified as a binary vector representing descending powers of the polynomial. In this case, the leading '1' of the polynomial must be included. It can also be specified as a string, prefaced by '0x', that is a hexadecimal representation of the descending powers of the polynomial. In this case, the leading '1' of the polynomial is omitted.
`InitialState`	The initial contents of the shift register. This property can be specified as a binary scalar, a binary vector, or as a string, prefaced by '0x', that is a hexadecimal representation of the binary vector. As a binary vector, its length must be one less than the length of the binary vector representation of the Polynomial.
`ReflectInput`	A Boolean quantity that specifies whether the input data should be flipped on a bytewise basis prior to entering the shift register.

Property	Description
`ReflectRemainder`	A Boolean quantity that specifies whether the binary output CRC checksum should be flipped around its center after the input data is completely through the shift register.
`FinalXOR`	The value with which the CRC checksum is to be XORed just prior to detecting the input data. This property can be specified as a binary scalar, a binary vector or as a string, prefaced by '0x', that is a hexadecimal representation of the binary vector. As a binary vector, its length must be one less than the length of the binary vector representation of the Polynomial.

A detect method is used with the object to detect errors in digital transmission.

CRC Generation Algorithm

For information pertaining to the CRC generation algorithm, see in the Communications System Toolbox User's Guide.

Detector Method

`[OUTDATA ERROR] = DETECT(H, INDATA)` detects transmission errors in the encoded input message `INDATA` by regenerating a CRC checksum using the CRC detector object H. The detector then compares the regenerated checksum with the checksum appended to INDATA. The binary-valued INDATA can be either a column vector or a matrix. If it is a matrix, each column is considered to be a separate channel. OUTDATA is identical to the input message INDATA, except that it has the CRC checksum stripped off. ERROR is a 1xC logical vector indicating if the encoded message INDATA has errors, where C is the number of channels in INDATA. An ERROR value of 0 indicates no errors, and a value of 1 indicates errors.

Examples

The following three examples demonstrate the use of constructing an object. The fourth example demonstrates use of the detect method.

```
% Construct a CRC detector with a polynomial
% defined by x^4+x^3+x^2+x+1:
h = crc.detector([1 1 1 1 1])
```

This example generates the following output:

```
h =

               Type: CRC Detector
         Polynomial: 0xF
       InitialState: 0x0
       ReflectInput: false
   ReflectRemainder: false
           FinalXOR: 0x0
```

```
% Construct a CRC detector with a polynomial
% defined by x^3+x+1, with
% zero initial states, and with an all-ones
% final XOR value:
h = crc.detector('Polynomial', [1 0 1 1], ...
'InitialState', [0 0 0], 'FinalXOR', [1 1 1])
```

This example generates the following output:

```
h =

               Type: CRC Detector
         Polynomial: [1 0 1 1]
       InitialState: [0 0 0]
       ReflectInput: false
   ReflectRemainder: false
```

```
            FinalXOR: [1 1 1]

% Construct a CRC detector with a polynomial
% defined by x^4+x^3+x^2+x+1,
% all-ones initial states, reflected input, and all-zeros
% final XOR value:
  h = crc.detector('Polynomial', '0xF', 'InitialState', ...
  '0xF', 'ReflectInput', true, 'FinalXOR', '0x0')
```

This example generates the following output:

```
h =

                Type: CRC Detector
          Polynomial: 0xF
        InitialState: 0xF
        ReflectInput: true
    ReflectRemainder: false
            FinalXOR: 0x0

% Create a CRC-16 CRC generator, then use it to generate
% a checksum for the
% binary vector represented by the
% ASCII sequence '123456789'.
% Introduce an error, then detect it
% using a CRC-16 CRC detector.
gen = crc.generator('Polynomial', '0x8005', 'ReflectInput', ...
true, 'ReflectRemainder', true);
det = crc.detector('Polynomial', '0x8005', 'ReflectInput', ...
true, 'ReflectRemainder', true);
% The message below is an ASCII representation
% of the digits 1-9
msg = reshape(de2bi(49:57, 8, 'left-msb')', 72, 1);
encoded = generate(gen, msg);
encoded(1) = ~encoded(1);                 % Introduce an error
[outdata error] = detect(det, encoded);  % Detect the error
noErrors = isequal(msg, outdata)         % Should be 0
error                                    % Should be 1
```

This example generates the following output:

```
noErrors =

     0


error =

     1
```

See Also

`crc.generator`

Purpose

Construct CRC generator object

Syntax

```
h = crc.generator(polynomial)
```

```
h = crc.generator(detectorObj)
```

```
h = crc.generator(`Polynomial', polynomial, `param1', val1,
etc.)
```

```
h = crc.generator
```

Description

`h = crc.generator(polynomial)` constructs a CRC generator object H defined by the generator polynomial POLYNOMIAL.

`h = crc.generator(detectorObj)` constructs a CRC generator object H defined by the parameters found in the CRC detector object DETECTOROBJ.

`h = crc.generator(`property1', val1, ...)` constructs a CRC generator object H with properties as specified by the PROPERTY/VALUE pairs.

`h = crc.generator` constructs a CRC generator object H with default properties. It constructs a CRC-CCITT generator, and is equivalent to: h = crc.generator('Polynomial', '0x1021', 'InitialState', '0xFFFF', ...

'ReflectInput', false, 'ReflectRemainder', false, 'FinalXOR', '0x0000').

Properties

The following table describes the properties of a CRC generator object. All properties are writable, except `Polynomial`.

Property	Description
`Polynomial`	The generator polynomial that defines connections for a linear feedback shift register. This property can be specified as a binary vector representing descending powers of the polynomial. In this case, the leading '1' of the polynomial must be included. It can also be specified as a string, prefaced by '0x', that is a hexadecimal representation of the descending powers of the polynomial. In this case, the leading '1' of the polynomial is omitted.
`InitialState`	The initial contents of the shift register. This property can be specified as a binary scalar, a binary vector, or as a string, prefaced by '0x', that is a hexadecimal representation of the binary vector. As a binary vector, its length must be one less than the length of the binary vector representation of the Polynomial.
`ReflectInput`	A Boolean quantity that specifies whether the input data should be flipped on a bytewise basis prior to entering the shift register.

Property	Description
`ReflectRemainder`	A Boolean quantity that specifies whether the binary output CRC checksum should be flipped around its center after the input data is completely through the shift register.
`FinalXOR`	The value with which the CRC checksum is to be XORed just prior to being appended to the input data. This property can be specified as a binary scalar, a binary vector, or as a string, prefaced by '0x', that is a hexadecimal representation of the binary vector. As a binary vector, its length must be one less than the length of the binary vector representation of the Polynomial.

CRC Generation Algorithm

For information pertaining to the CRC generation algorithm, refer to the "CRC Non-Direct Algorithm" section of the Communications System Toolbox User's Guide.

Generator Method

encoded = generate(h, msg) generates a CRC checksum for an input message using the CRC generator object H. It appends the checksum to the end of MSG. The binary-valued MSG can be either a column vector or a matrix. If it is a matrix, then each column is considered to be a separate channel.

Usage Example

The following examples demonstrate the use of this object.

```
% Construct a CRC generator with a polynomial defined
% by x^4+x^3+x^2+x+1:
h = crc.generator([1 1 1 1 1])

% Construct a CRC generator with a polynomial defined
% by x^4+x^3+x^2+x+1, all-ones initial states, reflected
% input, and all-zeros final XOR value:
h = crc.generator('Polynomial', '0xF', 'InitialState', ...
'0xF', 'ReflectInput', true, 'FinalXOR', '0x0')

% Create a CRC-16 CRC generator, then use it to generate
% a checksum for the
% binary vector represented by the ASCII sequence '123456789'.
gen = crc.generator('Polynomial', '0x8005', ...
'ReflectInput', true, 'ReflectRemainder', true);
% The message below is an ASCII representation of ...
% the digits 1-9
msg = reshape(de2bi(49:57, 8, 'left-msb')', 72, 1);
encoded = generate(gen, msg);

% Construct a CRC generator with a polynomial defined
% by x^3+x+1, with zero initial states,
% and with an all-ones final XOR value:
h = crc.generator('Polynomial', [1 0 1 1], ...
                  'InitialState', [0 0 0], ...
                  'FinalXOR', [1 1 1])
```

See Also

`crc.detector`

Purpose

Produce parity-check and generator matrices for cyclic code

Syntax

```
h = cyclgen(n,pol)
h = cyclgen(n,pol,opt)
[h,g] = cyclgen(...)
[h,g,k] = cyclgen(...)
```

Description

For all syntaxes, the codeword length is `n` and the message length is `k`. A polynomial can generate a cyclic code with codeword length `n` and message length `k` if and only if the polynomial is a degree-(`n-k`) divisor of x^`n`-1. (Over the binary field GF(2), x^`n`-1 is the same as x^`n`+1.) This implies that `k` equals `n` minus the degree of the generator polynomial.

`h = cyclgen(n,pol)` produces an (`n-k`)-by-`n` parity-check matrix for a systematic binary cyclic code having codeword length `n`. The row vector `pol` gives the binary coefficients, in order of ascending powers, of the degree-(`n-k`) generator polynomial.

`h = cyclgen(n,pol,`*`opt`*`)` is the same as the syntax above, except that the argument *`opt`* determines whether the matrix should be associated with a systematic or nonsystematic code. The values for *`opt`* are `'system'` and `'nonsys'`.

`[h,g] = cyclgen(...)` is the same as `h = cyclgen(...)`, except that it also produces the `k`-by-`n` generator matrix `g` that corresponds to the parity-check matrix `h`.

`[h,g,k] = cyclgen(...)` is the same as `[h,g] = cyclgen(...)`, except that it also returns the message length `k`.

Examples

The code below produces parity-check and generator matrices for a binary cyclic code with codeword length 7 and message length 4.

```
pol = cyclpoly(7,4);
[parmat,genmat,k] = cyclgen(7,pol)
```

The output is

```
parmat =
```

```
     1     0     0     1     1     1     0
     0     1     0     0     1     1     1
     0     0     1     1     1     0     1


genmat =

     1     0     1     1     0     0     0
     1     1     1     0     1     0     0
     1     1     0     0     0     1     0
     0     1     1     0     0     0     1


k =

     4
```

In the output below, notice that the parity-check matrix is different from `parmat` above, because it corresponds to a nonsystematic cyclic code. In particular, `parmatn` does not have a 3-by-3 identity matrix in its leftmost three columns, as `parmat` does.

```
parmatn = cyclgen(7,cyclpoly(7,4),'nonsys')
parmatn =

     1     1     1     0     1     0     0
     0     1     1     1     0     1     0
     0     0     1     1     1     0     1
```

See Also

`encode` | `decode` | `bchgenpoly` | `cyclpoly`

How To

- “Block Codes”

Purpose

Produce generator polynomials for cyclic code

Syntax

```
pol = cyclpoly(n,k)
pol = cyclpoly(n,k,opt)
```

Description

For all syntaxes, a polynomial is represented as a row containing the coefficients in order of ascending powers.

`pol = cyclpoly(n,k)` returns the row vector representing one nontrivial generator polynomial for a cyclic code having codeword length `n` and message length `k`.

`pol = cyclpoly(n,k,opt)` searches for one or more nontrivial generator polynomials for cyclic codes having codeword length `n` and message length `k`. The output `pol` depends on the argument *`opt`* as shown in the table below.

opt	Significance of pol	Format of pol
`'min'`	One generator polynomial having the smallest possible weight	Row vector representing the polynomial
`'max'`	One generator polynomial having the greatest possible weight	Row vector representing the polynomial
`'all'`	All generator polynomials M	Matrix, each row of which represents one such polynomial
a positive integer, L	All generator polynomials having weight L	Matrix, each row of which represents one such polynomial

The weight of a binary polynomial is the number of nonzero terms it has. If no generator polynomial satisfies the given conditions, the output `pol` is empty and a warning message is displayed.

Examples

The first command below produces representations of three generator polynomials for a [15,4] cyclic code. The second command shows that $1 + x + x^2 + x^3 + x^5 + x^7 + x^8 + x^{11}$ is one such polynomial having the largest number of nonzero terms.

```
c1 = cyclpoly(15,4,'all')
c2 = cyclpoly(15,4,'max')
```

The output is

```
c1 =

  Columns 1 through 10

     1     1     0     0     0     1     1     0     0     0
     1     0     0     1     1     0     1     0     1     1
     1     1     1     1     0     1     0     1     1     0

  Columns 11 through 12

     1     1
     1     1
     0     1


c2 =

  Columns 1 through 10

     1     1     1     1     0     1     0     1     1     0

  Columns 11 through 12
```

```
     0     1
```

This command shows that no generator polynomial for a [15,4] cyclic code has exactly three nonzero terms.

```
c3 = cyclpoly(15,4,3)

Warning: No cyclic generator polynomial satisfies the given constraints.
> In cyclpoly at 131

c3 =

     []
```

Algorithms

If *opt* is `'min'`, `'max'`, or omitted, polynomials are constructed by converting decimal integers to base `p`. Based on the decimal ordering, `gfprimfd` returns the first polynomial it finds that satisfies the appropriate conditions. This algorithm is similar to the one used in `gfprimfd`.

See Also

`cyclgen` | `encode`

How To

- “Block Codes”

Purpose

Convert decimal numbers to binary vectors

Syntax

```
b = de2bi(d)
b = de2bi(d,n)
b = de2bi(d,n,p)
b = de2bi(d,[],p)
b = de2bi(d,...,flg)
```

Description

`b = de2bi(d)` converts a nonnegative decimal integer `d` to a binary row vector. If `d` is a vector, the output `b` is a matrix, each row of which is the binary form of the corresponding element in `d`. If `d` is a matrix, `de2bi` treats it like the vector `d(:)`.

Note By default, `de2bi` uses the first column of `b` as the *lowest*-order digit.

`b = de2bi(d,n)` is the same as `b = de2bi(d)`, except that its output has `n` columns, where `n` is a positive integer. An error occurs if the binary representations would require more than `n` digits. If necessary, the binary representation of `d` is padded with extra zeros.

`b = de2bi(d,n,p)` converts a nonnegative decimal integer `d` to a base-`p` row vector, where `p` is an integer greater than or equal to 2. The first column of `b` is the *lowest* base-`p` digit. `b` is padded with extra zeros if necessary, so that it has `n` columns, where `n` is a positive integer. An error occurs if the base-`p` representations would require more than `n` digits. If `d` is a nonnegative decimal vector, the output `b` is a matrix, each row of which is the (possibly zero-padded) base-p form of the corresponding element in `d`. If `d` is a matrix, `de2bi` treats it like the vector `d(:)`.

`b = de2bi(d,[],p)` specifies the base `p` but not the number of columns.

`b = de2bi(d,...,`*`flg`*`)` uses the string *`flg`* to determine whether the first column of `b` contains the lowest-order or highest-order digits. Values for *flg* are `'right-msb'` and `'left-msb'`. The value `'right-msb'` produces the default behavior.

Examples

The code below counts to 10 in decimal and binary.

```
d = (1:10)';
b = de2bi(d);
disp('    Dec          Binary       ')
disp('   -----   -------------------')
disp([d, b])
```

The output is below.

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

The command below shows how `de2bi` pads its output with zeros.

```
bb = de2bi([3 9],5) % Zero-padding the output

bb =

     1      1      0      0      0
     1      0      0      1      0
```

The commands below show how to convert a decimal integer to base three without specifying the number of columns in the output matrix.

They also show how to place the most significant digit on the left instead of on the right.

```
t = de2bi(12,[],3) % Convert 12 to base 3.

tleft = de2bi(12,[],3,'left-msb') % Significant digit on left
```

The output is

```
t =

     0     1     1


tleft =

     1     1     0
```

See Also

bi2de

Purpose

Block decoder

Syntax

```
msg = decode(code,n,k,'hamming/fmt',prim_poly)
msg = decode(code,n,k,'linear/fmt',genmat,trt)
msg = decode(code,n,k,'cyclic/fmt',genpoly,trt)
msg = decode(code,n,k)
[msg,err] = decode(...)
[msg,err,ccode] = decode(...)
[msg,err,ccode,cerr] = decode(...)
```

Optional Inputs

Input	Default Value
fmt	binary
prim_poly	gfprimdf(m) where n = 2^m-1
genpoly	cyclpoly(n,k)
trt	Uses syndtable to create the syndrome decoding table associated with the method's parity-check matrix

Description

For All Syntaxes

The decode function aims to recover messages that were encoded using an error-correction coding technique. The technique and the defining parameters must match those that were used to encode the original signal.

The "For All Syntaxes" on page 1-234 section on the encode reference page explains the meanings of n and k, the possible values of *fmt*, and the possible formats for code and msg. You should be familiar with the conventions described there before reading the rest of this section. Using the decode function with an input argument code that was *not* created by the encode function might cause errors.

For Specific Syntaxes

`msg = decode(code,n,k,'hamming/fmt',prim_poly)` decodes `code` using the Hamming method. For this syntax, `n` must have the form 2^m-1 for some integer m greater than or equal to 3, and `k` must equal `n`-m. `prim_poly` is a row vector that gives the binary coefficients, in order of ascending powers, of the primitive polynomial for GF(2^m) that is used in the encoding process. The default value of `prim_poly` is `gfprimdf(m)`. The decoding table that the function uses to correct a single error in each codeword is `syndtable`(`hammgen`(m)).

`msg = decode(code,n,k,'linear/fmt',genmat,trt)` decodes `code`, which is a linear block code determined by the `k`-by-`n` generator matrix `genmat`. `genmat` is required as input. `decode` tries to correct errors using the decoding table `trt`, where `trt` is a `2^(n-k)`-by-`n` matrix.

`msg = decode(code,n,k,'cyclic/fmt',genpoly,trt)` decodes the cyclic code `code` and tries to correct errors using the decoding table `trt`, where `trt` is a `2^(n-k)`-by-`n` matrix. `genpoly` is a row vector that gives the coefficients, in order of ascending powers, of the binary generator polynomial of the code. The default value of `genpoly` is `cyclpoly(n,k)`. By definition, the generator polynomial for an [`n`, `k`] cyclic code must have degree `n-k` and must divide x^n-1.

`msg = decode(code,n,k)` is the same as `msg = decode(code,n,k,'hamming/binary')`.

`[msg,err] = decode(...)` returns a column vector `err` that gives information about error correction. If the code is a convolutional code, `err` contains the metric calculations used in the decoding decision process. For other types of codes, a nonnegative integer in the rth row of `err` indicates the number of errors corrected in the rth *message* word; a negative integer indicates that there are more errors in the rth word than can be corrected.

`[msg,err,ccode] = decode(...)` returns the corrected code in `ccode`.

`[msg,err,ccode,cerr] = decode(...)` returns a column vector `cerr` whose meaning depends on the format of `code`:

- If `code` is a binary vector, a nonnegative integer in the rth row of `vec2matcerr` indicates the number of errors corrected in the rth *codeword*; a negative integer indicates that there are more errors in the rth codeword than can be corrected.
- If `code` is not a binary vector, `cerr = err`.

Examples

On the reference page for `encode`, some of the example code illustrates the use of the `decode` function.

The example below illustrates the use of `err` and `cerr` when the coding method is not convolutional code and the code is a binary vector. The script encodes two five-bit messages using a cyclic code. Each codeword has 15 bits. Errors are added to the first two bits of the first codeword and the first bit of the second codeword. Then `decode` is used to recover the original message. As a result, the errors are corrected. `err` reflects the fact that the first *message* was recovered after correcting two errors, while the second message was recovered after correcting one error. `cerr` reflects the fact that the first *codeword* was decoded after correcting two errors, while the second codeword was decoded after correcting one error.

```
m = 4; n = 2^m-1; % Codeword length is 15.
k = 5; % Message length
msg = ones(10,1); % Two messages, five bits each
code = encode(msg,n,k,'cyclic');  % Encode the message.
% Now place two errors in first word and one error
% in the second word.  Create errors by reversing bits.
noisycode = code;
noisycode(1:2) = bitxor(noisycode(1:2),[1 1]');
noisycode(16) = bitxor(noisycode(16),1);
% Decode and try to correct the errors.
[newmsg,err,ccode,cerr] = decode(noisycode,n,k,'cyclic');
disp('Transpose of err is'); disp(err')
disp('Transpose of cerr is'); disp(cerr')
```

The output is below.

```
Single-error patterns loaded in decoding table.
 1008 rows remaining.
2-error patterns loaded.  918 rows remaining.
3-error patterns loaded.  648 rows remaining.
4-error patterns loaded.  243 rows remaining.
5-error patterns loaded.  0 rows remaining.
Transpose of err is
     2     1

Transpose of cerr is
     2     1
```

Algorithms

Depending on the decoding method, `decode` relies on such lower-level functions as `hammgen`, `syndtable`, and `cyclgen`.

See Also

`encode` | `cyclpoly` | `syndtable` | `gen2par`

How To

- "Block Codes"

deintrlv

Purpose

Restore ordering of symbols

Syntax

`deintrlvd = deintrlv(data,elements)`

Description

`deintrlvd = deintrlv(data,elements)` restores the original ordering of the elements of `data` by acting as an inverse of `intrlv`. If `data` is a length-N vector or an N-row matrix, `elements` is a length-N vector that permutes the integers from 1 to N. To use this function as an inverse of the `intrlv` function, use the same `elements` input in both functions. In that case, the two functions are inverses in the sense that applying `intrlv` followed by `deintrlv` leaves data unchanged.

Examples

The code below illustrates the inverse relationship between `intrlv` and `deintrlv`.

```
p = randperm(10); % Permutation vector
a = intrlv(10:10:100,p); % Rearrange [10 20 30 ... 100].
b = deintrlv(a,p) % Deinterleave a to restore ordering.
```

The output is

```
b =

    10    20    30    40    50    60    70    80    90   100
```

See Also

`intrlv`

How To

- “Interleaving”

Purpose

Construct decision-feedback equalizer object

Syntax

```
eqobj = dfe(nfwdweights,nfbkweights,alg)
eqobj = dfe(nfwdweights,nfbkweights,alg,sigconst)
eqobj = dfe(nfwdweights,nfbkweights,alg,sigconst,nsamp)
```

Description

The `dfe` function creates an equalizer object that you can use with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see "Adaptive Algorithms".

`eqobj = dfe(nfwdweights,nfbkweights,alg)` constructs a decision feedback equalizer object. The equalizer's feedforward and feedback filters have `nfwdweights` and `nfbkweights` symbol-spaced complex weights, respectively, which are initially all zeros. `alg` describes the adaptive algorithm that the equalizer uses; you should create `alg` using any of these functions: `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma`. The signal constellation of the desired output is `[-1 1]`, which corresponds to binary phase shift keying (BPSK).

`eqobj = dfe(nfwdweights,nfbkweights,alg,sigconst)` specifies the signal constellation vector of the desired output.

`eqobj = dfe(nfwdweights,nfbkweights,alg,sigconst,nsamp)` constructs a DFE with a fractionally spaced forward filter. The forward filter has `nfwdweights` complex weights spaced at `T/nsamp`, where `T` is the symbol period and `nsamp` is a positive integer. `nsamp = 1` corresponds to a symbol-spaced forward filter.

Properties

The table below describes the properties of the decision feedback equalizer object. To learn how to view or change the values of a decision feedback equalizer object, see "Accessing Properties of an Equalizer".

Note To initialize or reset the equalizer object `eqobj`, enter `reset(eqobj)`.

Property	Description
`EqType`	Fixed value, `'Decision Feedback Equalizer'`
`AlgType`	Name of the adaptive algorithm represented by `alg`
`nWeights`	Number of weights in the forward filter and the feedback filter, in the format `[nfwdweights, nfbkweights]`. The number of weights in the forward filter must be at least 1.
`nSampPerSym`	Number of input samples per symbol (equivalent to `nsamp` input argument). This value relates to both the equalizer structure (see the use of K in "Decision-Feedback Equalizers") and an assumption about the signal to be equalized.
`RefTap` (except for CMA equalizers)	Reference tap index, between 1 and `nfwdweights`. Setting this to a value greater than 1 effectively delays the reference signal with respect to the equalizer's input signal.
`SigConst`	Signal constellation, a vector whose length is typically a power of 2.

Property	Description
`Weights`	Vector that concatenates the complex coefficients from the forward filter and the feedback filter. This is the set of w_i values in the schematic in "Decision-Feedback Equalizers".
`WeightInputs`	Vector that concatenates the tap weight inputs for the forward filter and the feedback filter. This is the set of u_i values in the schematic in "Decision-Feedback Equalizers".
`ResetBeforeFiltering`	If `1`, each call to `equalize` resets the state of `eqobj` before equalizing. If `0`, the equalization process maintains continuity from one call to the next.
`NumSamplesProcessed`	Number of samples the equalizer processed since the last reset. When you create or reset `eqobj`, this property value is `0`.
Properties specific to the adaptive algorithm represented by `alg`	See reference page for the adaptive algorithm function that created `alg`: `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma`.

Relationships Among Properties

If you change `nWeights`, MATLAB maintains consistency in the equalizer object by adjusting the values of the properties listed below.

Property	Adjusted Value
`Weights`	`zeros(1,sum(nWeights))`
`WeightInputs`	`zeros(1,sum(nWeights))`
`StepSize` (Variable-step-size LMS equalizers)	`InitStep*ones(1,sum(nWeights))`
`InvCorrMatrix` (RLS equalizers)	`InvCorrInit*eye(sum(nWeights))`

An example illustrating relationships among properties is in "Linked Properties of an Equalizer Object".

Apply a decision feedback equalizer (DFE) to an 8-PSK modulated signal

This example shows how to apply a decision feedback equalizer (DFE) to an 8-PSK modulated signal impaired by a frequency selector channel. The DFE uses 400 training symbols.

Set the modulation order to define 8-PSK modulation, and create a PSK modulator System object™.

```
M = 8;
hMod = comm.PSKModulator(M);
```

Create a 1500-by-1 column vector of random message symbols.

```
msg = randi([0 M-1],1500,1);
```

Modulate the random message signal by calling the `step` method of the `comm.PSKModulator` System object.

```
modmsg = step(hMod,msg);
```

Define a frequency selective channel with four taps, and then pass the modulated signal through the channel, introducing channel distortion.

```
chan = [.986; .845; .237; .123+.31i];
filtmsg = filter(chan,1,modmsg);
```

Create a DFE equalizer that has 10 feed forward tabs and five feedback tabs. The equalizer uses the LMS update method with a step size of 0.01.

```
numFFTaps = 10; numFBTaps = 5;
eq1 = dfe(numFFTaps, numFBTaps, lms(0.01));
```

For decision directed operation, the DFE must use the same signal constellation as the transmission scheme. Set the SigConst property to the constellation the modulator System object uses.

```
eq1.SigConst = step(hMod,(0:M-1)')';
```

Equalize the signal to help remove the effects of channel distortion. Use the first 400 symbols to train the equalizer.

```
trainlen = 400;
[symbolest,yd] = equalize(eq1,filtmsg,modmsg(1:trainlen));
```

Plot the received signal, equalizer output after training, and the ideal signal constellation.

```
h = scatterplot(filtmsg,1,trainlen,'bx'); hold on;
scatterplot(symbolest,1,trainlen,'g.',h);
scatterplot(eq1.SigConst,1,0,'k*',h);
legend('Filtered signal','Equalized signal',...
   'Ideal signal constellation');
hold off;
```

Demodulate the signal at the equalizer output, and the unequalized signal at the input of the equalizer.

```
hDemod = comm.PSKDemodulator(8);
demodmsg_noeq = step(hDemod,filtmsg);
demodmsg = step(hDemod,yd);
```

Compute the error rates for the two demodulated signals and compare the results.

```
hErrorCalc = comm.ErrorRate;
ser_noEq = step(hErrorCalc, ...
    msg(trainlen+1:end), demodmsg_noeq(trainlen+1:end));
reset(hErrorCalc)
ser_Eq = step(hErrorCalc, msg(trainlen+1:end),demodmsg(trainlen+1:end));
disp('Symbol error rates with and without equalizer:')
disp([ser_Eq(1) ser_noEq(1)])
```

The equalizer helps eliminate the distortion introduced by the frequency selective channel, and reduces the error rate.

See Also

`lms` | `signlms` | `normlms` | `varlms` | `rls` | `cma` | `lineareq` | `equalize`

How To

- "Equalization"

Purpose

Discrete Fourier transform matrix in Galois field

Syntax

```
dm = dftmtx(alph)
```

Description

`dm = dftmtx(alph)` returns a Galois array that represents the discrete Fourier transform operation on a Galois vector, with respect to the Galois scalar `alph`. The element `alph` is a primitive nth root of unity in the Galois field $GF(2^m) = GF(n+1)$; that is, n must be the smallest positive value of `k` for which `alph^k` equals 1. The discrete Fourier transform has size n and `dm` is an n-by-n array. The array `dm` represents the transform in the sense that `dm` times any length-n Galois column vector yields the transform of that vector.

Note The inverse discrete Fourier transform matrix is `dftmtx(1/alph)`.

Examples

The example below illustrates the discrete Fourier transform and its inverse, with respect to the element `gf(3,4)`. The example examines the first n powers of that element to make sure that only the nth power equals one. Afterward, the example transforms a random Galois vector, undoes the transform, and checks the result.

```
m = 4;
n = 2^m-1;
a = 3;
alph = gf(a,m);
mp = minpol(alph);
if (mp(1)==1 && isprimitive(mp)) % Check that alph has order n.
    disp('alph is a primitive nth root of unity.')
    dm = dftmtx(alph);
    idm = dftmtx(1/alph);
    x = gf(randi([0 2^m-1],n,1),m);
    y = dm*x; % Transform x.
    z = idm*y; % Recover x.
    ck = isequal(x,z)
```

```
end
```

The output is

```
alph is a primitive nth root of unity.

ck =

     1
```

Limitations

The Galois field over which this function works must have 256 or fewer elements. In other words, `alph` must be a primitive nth root of unity in the Galois field GF(2^m), where m is an integer between 1 and 8.

Algorithms

The element `dm(a,b)` equals `alph^((a-1)*(b-1))`.

See Also

`fft` | `ifft`

How To

- "Signal Processing Operations in Galois Fields"

Purpose

Compute distance spectrum of convolutional code

Syntax

```
spect = distspec(trellis,n)
spect = distspec(trellis)
```

Description

`spect = distspec(trellis,n)` computes the free distance and the first `n` components of the weight and distance spectra of a linear convolutional code. Because convolutional codes do not have block boundaries, the weight spectrum and distance spectrum are semi-infinite and are most often approximated by the first few components. The input `trellis` is a valid MATLAB trellis structure, as described in "Trellis Description of a Convolutional Code". The output, `spect`, is a structure with these fields:

Field	Meaning
`spect.dfree`	Free distance of the code. This is the minimum number of errors in the encoded sequence required to create an error event.
`spect.weight`	A length-`n` vector that lists the total number of information bit errors in the error events enumerated in `spect.event`.
`spect.event`	A length-`n` vector that lists the number of error events for each distance between `spect.dfree` and `spect.dfree+n-1`. The vector represents the first `n` components of the distance spectrum.

`spect = distspec(trellis)` is the same as `spect = distspec(trellis,1)`.

Examples

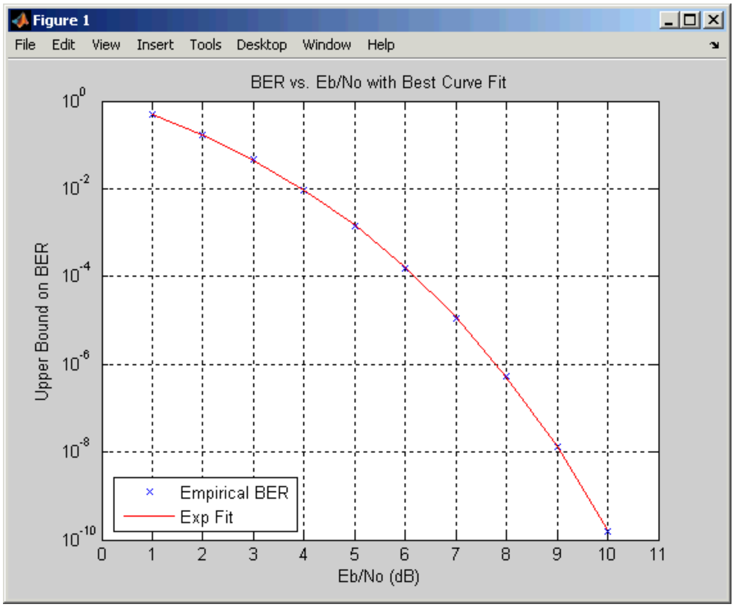
The example below performs these tasks:

- Computes the distance spectrum for the rate 2/3 convolutional code that is depicted on the reference page for the `poly2trellis` function
- Uses the output of `distspec` as an input to the `bercoding` function, to find a theoretical upper bound on the bit error rate for a system that uses this code with coherent BPSK modulation
- Plots the upper bound using the `berfit` function

```
trellis = poly2trellis([5 4],[23 35 0; 0 5 13])
spect = distspec(trellis,4)
berub = bercoding(1:10,'conv','hard',2/3,spect); % BER bound
berfit(1:10,berub); ylabel('Upper Bound on BER'); % Plot.
```

The output and plot are below.

```
trellis =

     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 128
          nextStates: [128x4 double]
             outputs: [128x4 double]


spect =

     dfree: 5
    weight: [1 6 28 142]
     event: [1 2 8 25]
```



Algorithms

The function uses a tree search algorithm implemented with a stack, as described in [2].

References

[1] Bocharova, I. E., and B. D. Kudryashov, "Rational Rate Punctured Convolutional Codes for Soft-Decision Viterbi Decoding," *IEEE Transactions on Information Theory*, Vol. 43, No. 4, July 1997, pp. 1305–1313.

[2] Cedervall, M., and R. Johannesson, "A Fast Algorithm for Computing Distance Spectrum of Convolutional Codes," *IEEE Transactions on Information Theory*, Vol. 35, No. 6, Nov. 1989, pp. 1146–1159.

[3] Chang, J., D. Hwang, and M. Lin, "Some Extended Results on the Search for Good Convolutional Codes," *IEEE Transactions on Information Theory*, Vol. 43, No. 5, Sep. 1997, pp. 1682–1697.

[4] Frenger, P., P. Orten, and T. Ottosson, “Comments and Additions to Recent Papers on New Convolutional Codes,” *IEEE Transactions on Information Theory*, Vol. 47, No. 3, March 2001, pp. 1199–1201.

See Also

`bercoding` | `iscatastrophic` | `istrellis` | `poly2trellis`

Purpose

Package of Doppler classes

Description

This package contains the classes that instantiate Doppler objects. These objects are used as values of the `DopplerSpectrum` property, which is common to both Rayleigh and Rician channel objects.

Properties and Methods

Every Doppler object has a read-only `SpectrumType` property. Other properties are specific to each Doppler class.

Every Doppler object has a `copy` method, to duplicate itself, and a `disp` method, to display its properties.

See Also

`doppler.ajakes` | `doppler.bell` | `doppler.bigaussian` | `doppler.flat` | `doppler.gaussian` | `doppler.jakes` | `doppler.rjakes` | `doppler.rounded` | `rayleighchan` | `ricianchan` | `stdchan`

How To

- "Fading Channels"

doppler.ajakes

Purpose

Construct asymmetrical Doppler spectrum object

Syntax

```
dop = doppler.ajakes(freqminmaxajakes)
dop = doppler.ajakes
```

Description

The `doppler.ajakes` function creates an asymmetrical Jakes (AJakes) Doppler spectrum object. This object is to be used for the `DopplerSpectrum` property of a channel object created with the `rayleighchan` or the `ricianchan` functions.

`dop = doppler.ajakes(freqminmaxajakes)`, where `freqminmaxajakes` is a row vector of two finite real numbers between -1 and 1, creates a Jakes Doppler spectrum that is nonzero only for normalized (by the maximum Doppler shift f_d, in Hz) frequencies f_{norm} such that $-1 \leq f_{\min,norm} \leq f_{norm} \leq f_{\max,norm} \leq 1$, where $f_{\min,norm}$ is given by `freqminmaxajakes(1)` and $f_{\max,norm}$ is given by `freqminmaxajakes(2)`. The maximum Doppler shift f_d is specified by the `MaxDopplerShift` property of the channel object. Analytically: $f_{\min,norm} = f_{\min} / f_d$ and $f_{\max,norm} = f_{\max} / f_d$, where $f_{\min}$ is the minimum Doppler shift (in hertz) and $f_{\max}$ is the maximum Doppler shift (in hertz).

When `dop` is used as the `DopplerSpectrum` property of a channel object, space `freqminmaxajakes(1)` and `freqminmaxajakes(2)` by more than 1/50. Assigning a smaller spacing results in `freqminmaxarjakes` being reset to the default value of `[0 1]`.

`dop = doppler.ajakes` creates an asymmetrical Doppler spectrum object with a default `freqminmaxajakes = [0 1]`. This syntax is equivalent to constructing a Jakes Doppler spectrum that is nonzero only for positive frequencies.

Properties

The AJakes Doppler spectrum object contains the following properties.

Property	Description
`SpectrumType`	Fixed value, `'AJakes'`
`FreqMinMaxAJakes`	Vector of minimum and maximum normalized Doppler shifts, two real finite numbers between -1 and 1

Theory and Applications

The Jakes power spectrum is based on the assumption that the angles of arrival at the mobile receiver are uniformly distributed [1]: the spectrum then covers the frequency range from $-f_d$ to f_d, f_d being the maximum Doppler shift. When the angles of arrival are not uniformly distributed, then the Jakes power spectrum does not cover the full Doppler bandwidth from $-f_d$ to f_d. The AJakes Doppler spectrum object covers the case of a power spectrum that is nonzero only for frequencies f such that $-f_d \le f_{\min} \le f \le f_{\max} \le f_d$. It is an asymmetrical spectrum in the general case, but becomes a symmetrical spectrum if $f_{\min} = -f_{\max}$.

The normalized AJakes Doppler power spectrum is given analytically by:

$$S(f) = \frac{A_a}{\pi f_d \sqrt{1-(f/f_d)^2}}, \quad -f_d \le f_{\min} \le f \le f_{\max} \le f_d$$

$$A_a = \frac{1}{\frac{1}{\pi}\left[\sin^{-1}\left(\frac{f_{\max}}{f_d}\right) - \sin^{-1}\left(\frac{f_{\min}}{f_d}\right)\right]}$$

where $f_{\min}$ and $f_{\max}$ denote the minimum and maximum frequencies where the spectrum is nonzero. You can determine these values from the probability density function of the angles of arrival.

Examples

The following MATLAB code first creates a Rayleigh channel object with a maximum Doppler shift of $f_d = 10$ Hz. It then creates an AJakes Doppler object with minimum normalized Doppler shift $f_{\min,norm} = -0.2$ and maximum normalized Doppler shift $f_{\max,norm} = 0.05$. The Doppler object is then assigned to the `DopplerSpectrum` property of the channel object. The channel then has a Doppler spectrum that is nonzero for frequencies f such that $-f_d \le f_{\min} \le f \le f_{\max} \le f_d$, where $f_{\min} = f_{\min,norm} \times f_d = -2$ Hz and $f_{\max} = f_{\max,norm} \times f_d = 0.5$ Hz.

```
chan = rayleighchan(1/1000, 10);
dop_ajakes = doppler.ajakes([-0.2 0.05]);
chan.DopplerSpectrum = dop_ajakes;
chan.DopplerSpectrum
```

This code returns:

```
         SpectrumType: 'AJakes'
    FreqMinMaxAJakes: [-0.2000 0.0500]
```

References

[1] Jakes, W. C., Ed., *Microwave Mobile Communications*, Wiley, 1974.

[2] Lee, W. C. Y., *Mobile Communications Engineering: Theory and Applications*, 2nd Ed., McGraw-Hill, 1998.

[3] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`doppler` | `doppler.bell` | `doppler.bigaussian` | `doppler.flat` | `doppler.gaussian` | `doppler.jakes` | `doppler.rjakes` | `doppler.rounded` | `rayleighchan` | `ricianchan` | `stdchan`

How To

- “Fading Channels”

doppler.bell

Purpose

Construct bell-shaped Doppler spectrum object

Syntax

```
doppler.bell
doppler.bell(coeffbell)
```

Description

doppler.bell creates a bell Doppler spectrum object. You can use this object with the `DopplerSpectrum` property of any channel object created with either the `rayleighchan` function, the `ricianchan` function, or `comm.MIMOChannel` System object.

dop = `doppler.bell` creates a bell Doppler spectrum object with default coefficient.

dop = `doppler.bell(coeffbell)` creates a bell Doppler spectrum object with coefficient given by `coeffbell`, where `coeffbell` is a positive, finite, real scalar.

Properties

The bell Doppler spectrum object has the following properties.

Property	Description
SpectrumType	Fixed value, 'Bell'
CoeffBell	Bell spectrum coefficient, positive real finite scalar.

Theory and Applications

A bell spectrum was proposed in [1] for the Doppler spectrum of indoor MIMO channels, for 802.11n channel modeling.

The normalized bell Doppler spectrum is given analytically by:

$$S(f) = \frac{C_b}{1+A\left(\frac{f}{f_d}\right)^2}$$

where

$$|f| \leq f_d$$

and

$$C_b = \frac{\sqrt{A}}{\pi f_d}$$

f_d represents the maximum Doppler shift specified for the channel object, and A represents a positive real finite scalar (`CoeffBell`). The indoor MIMO channel model of IEEE 802.11n [1] uses the following parameter: $A = 9$. Since the channel is modeled as Rician fading with a fixed line-of-sight (LOS) component, a Dirac delta is also present in the Doppler spectrum at $f = 0$.

Examples

Construct a bell Doppler spectrum object with a coefficient of 8.5. Assign it to a Rayleigh channel object with one path.

```
dop = doppler.bell(8.5);
chan = rayleighchan(1e-5, 10);
chan.DopplerSpectrum = dop;
```

References

[1] IEEE P802.11 Wireless LANs, "TGn Channel Models", IEEE 802.1103/940r4, 2004-05-10.

See Also

`doppler` | `doppler.ajakes` | `doppler.flat` | `doppler.gaussian` | `doppler.jakes` | `doppler.rjakes` | `doppler.rounded` | `rayleighchan` | `ricianchan` | `stdchan`

How To

- "Fading Channels"

doppler.bigaussian

Purpose Construct bi-Gaussian Doppler spectrum object

Syntax

```
dop = doppler.bigaussian(property1,value1,...)
dop = doppler.bigaussian
```

Description The `doppler.bigaussian` function creates a bi-Gaussian Doppler spectrum object to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` function or the `ricianchan` function).

`dop = doppler.bigaussian(property1,value1,...)` creates a bi-Gaussian Doppler spectrum object with properties as specified by the property/value pairs. If you do not specify a value for a property, the property is assigned a default value.

`dop = doppler.bigaussian` creates a bi-Gaussian Doppler spectrum object with default properties. The constructed Doppler spectrum object is equivalent to a single Gaussian Doppler spectrum centered at zero frequency. The equivalent command with property/value pairs is:

```
dop = doppler.bigaussian('SigmaGaussian1', 1/sqrt(2), ...
   'SigmaGaussian2', 1/sqrt(2), ...
   'CenterFreqGaussian1', 0, ...
   'CenterFreqGaussian2', 0, ...
   'GainGaussian1', 0.5, ...
   'GainGaussian2', 0.5)
```

Properties The bi-Gaussian Doppler spectrum object contains the following properties.

Property	Description
`SpectrumType`	Fixed value, `'BiGaussian'`
`SigmaGaussian1`	Normalized standard deviation of first Gaussian function (real positive finite scalar value)

Property	Description
`SigmaGaussian2`	Normalized standard deviation of second Gaussian function (real positive finite scalar value)
`CenterFreqGaussian1`	Normalized center frequency of first Gaussian function (real scalar value between -1 and 1)
`CenterFreqGaussian2`	Normalized center frequency of second Gaussian function (real scalar value between -1 and 1)
`GainGaussian1`	Power gain of first Gaussian function (linear scale, real nonnegative finite scalar value)
`GainGaussian2`	Power gain of second Gaussian function (linear scale, real nonnegative finite scalar value)

All properties are writable except for the `SpectrumType` property.

The properties `SigmaGaussian1`, `SigmaGaussian2`, `GainGaussian1`, and `GainGaussian2` are normalized by the `MaxDopplerShift` property of the associated channel object.

Analytically, the normalized standard deviations of the first and second Gaussian functions are determined as $\sigma_{G1,norm} = \sigma_{G1} / f_d$ and $\sigma_{G2,norm} = \sigma_{G2} / f_d$, respectively, where σ_{G1} and σ_{G2} are the standard deviations of the first and second Gaussian functions, and f_d is the maximum Doppler shift, in hertz. Similarly, the normalized center frequencies of the first and second Gaussian functions are determined as $f_{G1,norm} = f_{G1} / f_d$ and $f_{G2,norm} = f_{G2} / f_d$, respectively, where f_{G1} and f_{G2} are the center frequencies of the first and second Gaussian functions. The properties `GainGaussian1` and `GainGaussian2` correspond to the power gains C_{G1} and C_{G2}, respectively, of the two Gaussian functions.

Theory and Applications

The bi-Gaussian power spectrum consists of two frequency-shifted Gaussian spectra. The COST207 channel models ([1], [2], [3]) specify two distinct bi-Gaussian Doppler spectra, GAUS1 and GAUS2, to be used in modeling long echos for urban and hilly terrain profiles.

The normalized bi-Gaussian Doppler spectrum is given analytically by:

$$S_G(f) = A_G\left[\frac{C_{G1}}{\sqrt{2\pi\sigma_{G1}^2}}\exp\left(-\frac{(f-f_{G1})^2}{2\sigma_{G1}^2}\right)+\frac{C_{G,2}}{\sqrt{2\pi\sigma_{G2}^2}}\exp\left(-\frac{(f-f_{G2})^2}{2\sigma_{G2}^2}\right)\right]$$

where σ_{G1} and σ_{G2} are standard deviations, f_{G1} and f_{G2} are center frequencies, C_{G1} and C_{G2} are power gains, and $A_G = \frac{1}{C_{G1}+C_{G2}}$ is a normalization coefficient.

If either $f_{G1} = 0$ or $f_{G2} = 0$, a frequency-shifted Gaussian Doppler spectrum is obtained.

Examples

The following MATLAB code first creates a bi-Gaussian Doppler spectrum object with the same parameters as that of a COST 207 GAUS2 Doppler spectrum. It then creates a Rayleigh channel object with a maximum Doppler shift of $f_d = 30$ and assigns the constructed Doppler spectrum object to its `DopplerSpectrum` property.

```
dop_bigaussian = doppler.bigaussian('SigmaGaussian1', 0.1, ...
    'SigmaGaussian2', 0.15, 'CenterFreqGaussian1', 0.7, ...
    'CenterFreqGaussian2', -0.4, 'GainGaussian1', 1, ...
    'GainGaussian2', 1/10^1.5)
chan = rayleighchan(1e-3, 30);
chan.DopplerSpectrum = dop_bigaussian;
```

References

[1] COST 207 WG1, *Proposal on channel transfer functions to be used in GSM tests late 1986*, COST 207 TD (86) 51 Rev. 3, Sept. 1986.

[2] COST 207, *Digital land mobile radio communications*, Office for Official Publications of the European Communities, Final report, Luxembourg, 1989.

[3] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`doppler` | `doppler.ajakes` | `doppler.bell` | `doppler.flat` | `doppler.gaussian` | `doppler.jakes` | `doppler.rjakes` | `doppler.rounded` | `rayleighchan` | `ricianchan` | `stdchan`

How To

- “Fading Channels”

doppler.flat

Purpose

Construct flat Doppler spectrum object

Syntax

```
dop = doppler.flat
```

Description

`dop = doppler.flat` creates a flat Doppler spectrum object that is to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function). The maximum Doppler shift of the flat Doppler spectrum object is specified by the `MaxDopplerShift` property of the channel object.

Properties

The flat Doppler spectrum object contains only one property, `SpectrumType`, which is read-only and has a fixed value of `'Flat'`.

Theory and Applications

In a 3-D isotropic scattering environment, where the angles of arrival are uniformly distributed in the azimuth and elevation planes, the Doppler spectrum is found theoretically to be flat [2]. A flat Doppler spectrum is also specified in some cases of the ANSI J-STD-008 reference channel models for PCS, for both outdoor (pedestrian) and indoor (commercial) [1] applications.

The normalized flat Doppler power spectrum is given analytically by:

$$S(f) = \frac{1}{2f_d},\ |f| \le f_d$$

where f_d is the maximum Doppler frequency.

References

[1] ANSI J-STD-008, *Personal Station-Base Station Compatibility Requirements for 1.8 to 2.0 GHz Code Division Multiple Access (CDMA) Personal Communications Systems*, March 1995.

[2] Clarke, R. H., and Khoo, W. L., “3-D Mobile Radio Channel Statistics”, *IEEE Trans. Veh. Technol.*, Vol. 46, No. 3, pp. 798–799, August 1997.

See Also

`doppler` | `doppler.ajakes` | `doppler.bell` | `doppler.bigaussian` | `doppler.gaussian` | `doppler.jakes` | `doppler.rjakes` | `doppler.rounded` | `rayleighchan` | `ricianchan` | `stdchan`

How To

- "Fading Channels"

doppler.gaussian

Purpose

Construct Gaussian Doppler spectrum object

Syntax

```
dop = doppler.gaussian
dop = doppler.gaussian(sigmagaussian)
```

Description

The `doppler.gaussian` function creates a Gaussian Doppler spectrum object that is to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function).

`dop = doppler.gaussian` creates a Gaussian Doppler spectrum object with a default standard deviation (normalized by the maximum Doppler shift f_d, in Hz) $\sigma_{G,norm} = 1/\sqrt{2}$. The maximum Doppler shift f_d is specified by the `MaxDopplerShift` property of the channel object. Analytically, $\sigma_{G,norm} = \sigma_G / f_d = 1/\sqrt{2}$, where σ_G is the standard deviation of the Gaussian Doppler spectrum.

`dop = doppler.gaussian(sigmagaussian)` creates a Gaussian Doppler spectrum object with a normalized f_d (by the maximum Doppler shift f_d, in Hz) $\sigma_{G,norm}$ of value `sigmagaussian`.

Properties

The Gaussian Doppler spectrum object contains the following properties.

Property	Description
`SpectrumType`	Fixed value, `'Gaussian'`
`SigmaGaussian`	Normalized standard deviation of the Gaussian Doppler spectrum (a real positive number)

Theory and Applications

The Gaussian power spectrum is considered to be a good model for multipath components with long delays in UHF communications [3]. It is also proposed as a model for the aeronautical channel [2]. A Gaussian Doppler spectrum is also specified in some cases of the ANSI J-STD-008 reference channel models for PCS applications, for both outdoor (wireless loop) and indoor (residential, office) [1]. The normalized Gaussian Doppler power spectrum is given analytically by:

$$S_G(f) = \frac{1}{\sqrt{2\pi\sigma_G^2}} \exp\left(-\frac{f^2}{2\sigma_G^2}\right)$$

An alternate representation is [4]:

$$S_G(f) = \frac{1}{f_c}\sqrt{\frac{\ln 2}{\pi}} \exp\left(-(\ln 2)\left(\frac{f}{f_c}\right)^2\right)$$

where $f_c = \sigma_G \sqrt{2\ln 2}$ is the 3 dB cutoff frequency. If you set $f_c = f_d \sqrt{\ln 2}$, where f_d is the maximum Doppler shift, or equivalently $\sigma_G = f_d / \sqrt{2}$, the Doppler spread of the Gaussian power spectrum becomes equal to the Doppler spread of the Jakes power spectrum, where Doppler spread is defined as:

$$\sigma_D = \sqrt{\frac{\int_{-\infty}^{\infty} f^2 S(f) df}{\int_{-\infty}^{\infty} S(f) df}}$$

Examples

The following code creates a Rayleigh channel object with a maximum Doppler shift of $f_d = 10$. It then creates a Gaussian Doppler spectrum object with a normalized standard deviation of $\sigma_{G,\mathrm{norm}} = 0.5$, and assigns it to the `DopplerSpectrum` property of the channel object.

```
chan = rayleighchan(1/1000,10);
dop_gaussian = doppler.gaussian(0.5);
chan.DopplerSpectrum = dop_gaussian;
```

References

[1] ANSI J-STD-008, *Personal Station-Base Station Compatibility Requirements for 1.8 to 2.0 GHz Code Division Multiple Access (CDMA) Personal Communications Systems*, March 1995.

[2] Bello, P. A., "Aeronautical channel characterizations," *IEEE Trans. Commun.*, Vol. 21, pp. 548–563, May 1973.

[3] Cox, D. C., "Delay Doppler characteristics of multipath propagation at 910 MHz in a suburban mobile radio environment," *IEEE Transactions on Antennas and Propagation*, Vol. AP-20, No. 5, pp. 625–635, Sept. 1972.

[4] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`doppler` | `doppler.ajakes` | `doppler.bell` | `doppler.bigaussian` | `doppler.flat` | `doppler.jakes` | `doppler.rjakes` | `doppler.rounded` | `rayleighchan` | `ricianchan` | `stdchan`

How To

- "Fading Channels"

Purpose

Construct Jakes Doppler spectrum object

Syntax

Description

`dop = doppler.jakes` creates a Jakes Doppler spectrum object that is to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function). The maximum Doppler shift of the Jakes Doppler spectrum object is specified by the `MaxDopplerShift` property of the channel object. By default, channel objects are created with a Jakes Doppler spectrum.

Properties

The Jakes Doppler spectrum object contains only one property, `SpectrumType`, which is read-only and has a fixed value of `'Jakes'`.

Theory and Applications

The Jakes Doppler power spectrum model is actually due to Gans [2], who analyzed the Clarke-Gilbert model ([1], [3], and [5]). The Clarke-Gilbert model is also called the *classical model*.

The Jakes Doppler power spectrum applies to a mobile receiver. It derives from the following assumptions [6]:

- The radio waves propagate horizontally.
- At the mobile receiver, the angles of arrival of the radio waves are uniformly distributed over $[-\pi, \pi]$.
- At the mobile receiver, the antenna is omnidirectional (i.e., the antenna pattern is circular-symmetrical).

The normalized Jakes Doppler power spectrum is given analytically by:

$$S(f) = \frac{1}{\pi f_d \sqrt{1-(f/f_d)^2}}, \ |f| \le f_d$$

where f_d is the maximum Doppler frequency.

Examples

Create a Rayleigh channel object with a maximum Doppler shift of fd=10 Hertz. Then, create a Jakes Doppler spectrum object and assigns it to the `DopplerSpectrum` property of the channel object.

```
chan = rayleighchan(1/1000,10);
dop_gaussian = doppler.jakes;
chan.DopplerSpectrum = dop_gaussian
```

References

[1] Clarke, R. H., "A Statistical Theory of Mobile-Radio Reception," *Bell System Technical Journal*, Vol. 47, No. 6, pp. 957–1000, July-August 1968.

[2] Gans, M. J., "A Power-Spectral Theory of Propagation in the Mobile-Radio Environment," *IEEE Trans. Veh. Technol.*, Vol. VT-21, No. 1, pp. 27–38, Feb. 1972.

[3] Gilbert, E. N., "Energy Reception for Mobile Radio," *Bell System Technical Journal*, Vol. 44, No. 8, pp. 1779–1803, Oct. 1965.

[4] Jakes, W. C., Ed. *Microwave Mobile Communications*, Wiley, 1974.

[5] Lee, W. C. Y., *Mobile Communications Engineering: Theory and Applications*, 2nd Ed., McGraw-Hill, 1998.

[6] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`doppler` | `doppler.ajakes` | `doppler.bell` | `doppler.bigaussian` | `doppler.flat` | `doppler.gaussian` | `doppler.rjakes` | `doppler.rounded` | `rayleighchan` | `ricianchan` | `stdchan`

How To

- "Fading Channels"

Purpose

Construct restricted Jakes Doppler spectrum object

Syntax

```
dop = doppler.rjakes
dop = doppler.rjakes(freqminmaxrjakes)
```

Description

The `doppler.rjakes` function creates a restricted Jakes (RJakes) Doppler spectrum object that is used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function).

`dop = doppler.rjakes` creates a Doppler spectrum object equivalent to the Jakes Doppler spectrum. The maximum Doppler shift of the RJakes Doppler spectrum object is specified by the `MaxDopplerShift` property of the channel object.

`dop = doppler.rjakes(freqminmaxrjakes)`, where `freqminmaxrjakes` is a row vector of two finite real numbers between 0 and 1, creates a Jakes Doppler spectrum. This spectrum is nonzero only for normalized frequencies (by the maximum Doppler shift, f_d, in Hertz), f_{norm}, such that $0 \le f_{\min,norm} \le |f_{norm}| \le f_{\max,norm} \le 1$, where $f_{\min,norm}$ is given by `freqminmaxrjakes(1)` and $f_{\max,norm}$ is given by `freqminmaxrjakes(2)`. The maximum Doppler shift f_d is specified by the `MaxDopplerShift` property of the channel object.

Analytically, $f_{\min,norm} = f_{\min} / f_d$ and $f_{\max,norm} = f_{\max} / f_d$, where $f_{\min}$ is the minimum Doppler shift (in Hertz) and $f_{\max}$ is the maximum Doppler shift (in Hertz).

When `dop` is used as the `DopplerSpectrum` property of a channel object, `freqminmaxrjakes(1)` and `freqminmaxrjakes(2)` should be spaced by more than 1/50. Assigning a smaller spacing results in `freqminmaxrjakes` being reset to the default value of `[0 1]`.

Properties

The RJakes Doppler spectrum object contains the following properties.

Property	Description
`SpectrumType`	Fixed value, `'RJakes'`
`FreqMinMaxRJakes`	Vector of minimum and maximum normalized Doppler shifts (two real finite numbers between 0 and 1)

Theory and Applications

The Jakes power spectrum is based on the assumption that the angles of arrival at the mobile receiver are uniformly distributed [1], where the spectrum covers the frequency range from $-f_d$ to f_d, f_d being the maximum Doppler shift. When the angles of arrival are not uniformly distributed, the Jakes power spectrum does not cover the full Doppler bandwidth from $-f_d$ to f_d. This exception also applies to the case where the antenna pattern is directional. This type of spectrum is known as *restricted Jakes* [3]. The RJakes Doppler spectrum object covers only the case of a symmetrical power spectrum, which is nonzero only for frequencies f such that $0 \le f_{\min} \le |f| \le f_{\max} \le f_d$.

The normalized RJakes Doppler power spectrum is given analytically by:

$$S(f) = \frac{A_r}{\pi f_d \sqrt{1-(f/f_d)^2}},\ 0 \le f_{\min} \le |f| \le f_{\max} \le f_d$$

where

$$A_r = \frac{1}{\frac{2}{\pi}\left[\sin^{-1}\left(\frac{f_{\max}}{f_d}\right) - \sin^{-1}\left(\frac{f_{\min}}{f_d}\right)\right]}$$

$f_{\min}$ and $f_{\max}$ denote the minimum and maximum frequencies where the spectrum is nonzero. They can be determined from the probability density function of the angles of arrival.

Examples

The following code first creates a Rayleigh channel object with a maximum Doppler shift of $f_d = 10$. It then creates an RJakes Doppler object with minimum normalized Doppler shift $f_{\min,norm} = 0.14$ and maximum normalized Doppler shift $f_{\max,norm} = 0.9$.

The Doppler object is assigned to the `DopplerSpectrum` property of the channel object. The channel then has a Doppler spectrum that is nonzero for frequencies *f* such that $0 \le f_{\min} \le |f| \le f_{\max} \le f_d$, where $f_{\min} = f_{\min,norm} \times f_d = 1.4\text{ Hz}$ and $f_{\max} = f_{\max,norm} \times f_d = 9\text{ Hz}$.

```
chan = rayleighchan(1/1000, 10);
dop_rjakes = doppler.rjakes([0.14 0.9]);
chan.DopplerSpectrum = dop_rjakes;
chan.DopplerSpectrum
```

The output is:

```
         SpectrumType: 'RJakes'
    FreqMinMaxRJakes: [0.1400 0.9000]
```

References

[1] Jakes, W. C., Ed. *Microwave Mobile Communications*, Wiley, 1974.

[2] Lee, W. C. Y., *Mobile Communications Engineering: Theory and Applications*, 2nd Ed., McGraw-Hill, 1998.

[3] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`doppler` | `doppler.ajakes` | `doppler.bell` | `doppler.bigaussian` | `doppler.flat` | `doppler.gaussian` | `doppler.jakes` | `doppler.rounded` | `rayleighchan` | `ricianchan` | `stdchan`

How To

- “Fading Channels”

Purpose

Construct rounded Doppler spectrum object

Syntax

```
dop = doppler.rounded
dop = doppler.rounded(coeffrounded)
```

Description

The `doppler.rounded` function creates a rounded Doppler spectrum object that is used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function).

`dop = doppler.rounded` creates a rounded Doppler spectrum object with default polynomial coefficients $a_0 = 1$, $a_2 = -1.72$, $a_4 = 0.785$ (see "Theory and Applications" on page 1-221 for the meaning of these coefficients). The maximum Doppler shift f_d (in Hertz) is specified by the `MaxDopplerShift` property of the channel object.

`dop = doppler.rounded(coeffrounded)`, where `coeffrounded` is a row vector of three finite real numbers, creates a rounded Doppler spectrum object with polynomial coefficients, a_0, a_2, a_4, given by `coeffrounded(1)`, `coeffrounded(2)`, and `coeffrounded(3)`, respectively.

Properties

The rounded Doppler spectrum object contains the following properties.

Property	Description
`SpectrumType`	Fixed value, `'Rounded'`
`CoeffRounded`	Vector of three polynomial coefficients (real finite numbers)

Theory and Applications

A rounded spectrum is proposed as an approximation to the measured Doppler spectrum of the scatter component of fixed wireless channels at 2.5 GHz [1]. However, the shape of the spectrum is influenced by the center carrier frequency.

The normalized rounded Doppler spectrum is given analytically by a polynomial in f of order four, where only the even powers of f are retained:

$$S(f) = C_r\left[a_0 + a_2\left(\frac{f}{f_d}\right)^2 + a_4\left(\frac{f}{f_d}\right)^4\right],\ |f| \le f_d$$

where

$$C_r = \frac{1}{2f_d\left[a_0 + \frac{a_2}{3} + \frac{a_4}{5}\right]}$$

f_d is the maximum Doppler shift, and a_0, a_2, a_4 are real finite coefficients. The fixed wireless channel model of IEEE 802.16 [1] uses the following parameters: $a_0 = 1$, $a_2 = -1.72$, and $a_4 = 0.785$. Because the channel is modeled as Rician fading with a fixed line-of-sight (LOS) component, a Dirac delta is also present in the Doppler spectrum at $f = 0$.

Examples

The following code creates a Rician channel object with a maximum Doppler shift of $f_d = 10$. It then creates a rounded Doppler spectrum object with polynomial coefficients $a_0 = 1.0$, $a_2 = -0.5$, $a_4 = 1.5$, and assigns it to the `DopplerSpectrum` property of the channel object.

```
chan = ricianchan(1/1000,10,1);
dop_rounded = doppler.rounded([1.0 -0.5 1.5]);
chan.DopplerSpectrum = dop_rounded;
```

References

[1] IEEE 802.16 Broadband Wireless Access Working Group, “Channel models for fixed wireless applications,” *IEEE 802.16a-03/01*, 2003-06-27.

See Also

`doppler` | `doppler.ajakes` | `doppler.bell` | `doppler.bigaussian` | `doppler.flat` | `doppler.gaussian` | `doppler.jakes` | `doppler.rjakes` | `rayleighchan` | `ricianchan` | `stdchan`

How To

- "Fading Channels"

dpcmdeco

Purpose

Decode using differential pulse code modulation

Syntax

```
sig = dpcmdeco(indx,codebook,predictor)
[sig,quanterror] = dpcmdeco(indx,codebook,predictor)
```

Description

`sig = dpcmdeco(indx,codebook,predictor)` implements differential pulse code demodulation to decode the vector `indx`. The vector `codebook` represents the predictive-error quantization codebook. The vector `predictor` specifies the predictive transfer function. If the transfer function has predictive order M, `predictor` has length M+1 and an initial entry of 0. To decode correctly, use the same codebook and predictor in `dpcmenco` and `dpcmdeco`.

See "Represent Partitions", "Represent Codebooks", or the `quantiz` reference page, for a description of the formats of `partition` and `codebook`.

`[sig,quanterror] = dpcmdeco(indx,codebook,predictor)` is the same as the syntax above, except that the vector `quanterror` is the quantization of the predictive error based on the quantization parameters. `quanterror` is the same size as `sig`.

Note You can estimate the input parameters `codebook`, `partition`, and `predictor` using the function `dpcmopt`.

Examples

See "Example: DPCM Encoding and Decoding" and "Example: Comparing Optimized and Nonoptimized DPCM Parameters" for examples that use `dpcmdeco`.

References

[1] Kondoz, A. M., *Digital Speech*, Chichester, England, John Wiley & Sons, 1994.

See Also

`quantiz` | `dpcmopt` | `dpcmenco`

How To

- "Differential Pulse Code Modulation"

Purpose

Encode using differential pulse code modulation

Syntax

```
indx = dpcmenco(sig,codebook,partition,predictor)
[indx,quants] = dpcmenco(sig,codebook,partition,predictor)
```

Description

`indx = dpcmenco(sig,codebook,partition,predictor)` implements differential pulse code modulation to encode the vector `sig`. `partition` is a vector whose entries give the endpoints of the partition intervals. `codebook`, a vector whose length exceeds the length of `partition` by one, prescribes a value for each partition in the quantization. `predictor` specifies the predictive transfer function. If the transfer function has predictive order M, `predictor` has length M+1 and an initial entry of 0. The output vector `indx` is the quantization index.

See "Differential Pulse Code Modulation" for more about the format of `predictor`. See "Represent Partitions", "Represent Partitions", or the reference page for `quantiz` in this chapter, for a description of the formats of `partition` and `codebook`.

`[indx,quants] = dpcmenco(sig,codebook,partition,predictor)` is the same as the syntax above, except that `quants` contains the quantization of `sig` based on the quantization parameters. `quants` is a vector of the same size as `sig`.

Note If `predictor` is an order-one transfer function, the modulation is called a *delta modulation*.

Examples

See "Example: DPCM Encoding and Decoding" and "Example: Comparing Optimized and Nonoptimized DPCM Parameters" for examples that use `dpcmenco`.

References

[1] Kondoz, A. M., *Digital Speech*, Chichester, England, John Wiley & Sons, 1994.

See Also

`quantiz` | `dpcmopt` | `dpcmdeco`

How To

- "Differential Pulse Code Modulation"

Purpose

Optimize differential pulse code modulation parameters

Syntax

```
predictor = dpcmopt(training_set,ord)
[predictor,codebook,partition] = dpcmopt(training_set,ord,
   len)
[predictor,codebook,partition] =  dpcmopt(training_set,ord,
   ini_cb)
```

Description

`predictor = dpcmopt(training_set,ord)` returns a vector representing a predictive transfer function of order `ord` that is appropriate for the training data in the vector `training_set`. `predictor` is a row vector of length `ord+1`. See "Represent Predictors" for more about its format.

Note `dpcmopt` optimizes for the data in `training_set`. For best results, `training_set` should be similar to the data that you plan to quantize.

`[predictor,codebook,partition] = dpcmopt(training_set,ord,len)` is the same as the syntax above, except that it also returns corresponding optimized codebook and partition vectors `codebook` and `partition`. `len` is an integer that prescribes the length of `codebook`. `partition` is a vector of length `len`-1. See "Represent Partitions", "Represent Codebooks", or the reference page for `quantiz` in this chapter, for a description of the formats of `partition` and `codebook`.

`[predictor,codebook,partition] = dpcmopt(training_set,ord,ini_cb)` is the same as the first syntax, except that it also returns corresponding optimized codebook and partition vectors `codebook` and `partition`. `ini_cb`, a vector of length at least 2, is the initial guess of the codebook values. The output `codebook` is a vector of the same length as `ini_cb`. The output `partition` is a vector whose length is one less than the length of `codebook`.

Examples

See “Example: Comparing Optimized and Nonoptimized DPCM Parameters” for an example that uses `dpcmopt`.

See Also

`dpcmenco` | `dpcmdeco` | `quantiz` | `lloyds`

How To

- “Differential Pulse Code Modulation”

Purpose

Differential phase shift keying demodulation

Syntax

```
z = dpskdemod(y,M)
z = dpskdemod(y,M,phaserot)
z = dpskdemod(y,M,phaserot,symbol_order)
```

Description

`z = dpskdemod(y,M)` demodulates the complex envelope `y` of a DPSK modulated signal. `M` is the alphabet size and must be an integer. If `y` is a matrix with multiple rows and columns, the function processes the columns independently.

Note The first element of the output `z`, or the first row of `z`, if `z` is a matrix with multiple rows, represents an initial condition, because the differential algorithm compares two successive elements of the modulated signal.

`z = dpskdemod(y,M,phaserot)` specifies the phase rotation of the modulation in radians. In this case, the total phase shift per symbol is the sum of `phaserot` and the phase generated by the differential modulation.

`z = dpskdemod(y,M,phaserot,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray-coded ordering.

Examples

The example below illustrates the fact that the first output symbol of a differential PSK demodulator is an initial condition rather than useful information.

The output is below.

```
s1 =
```

```
     1


s2 =

     0
```

For another example that uses this function, see “Example: Curve Fitting for an Error Rate Plot”.

See Also

dpskmod | pskdemod | pskmod

How To

- “Digital Modulation”

Purpose

Differential phase shift keying modulation

Syntax

```
y = dpskmod(x,M)
y = dpskmod(x,M,phaserot)
y = dpskmod(x,M,phaserot,symbol_order)
```

Description

`y = dpskmod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using differential phase shift keying modulation. `M` is the alphabet size and must be an integer. The message signal must consist of integers between 0 and `M-1`. If `x` is a matrix with multiple rows and columns, the function processes the columns independently.

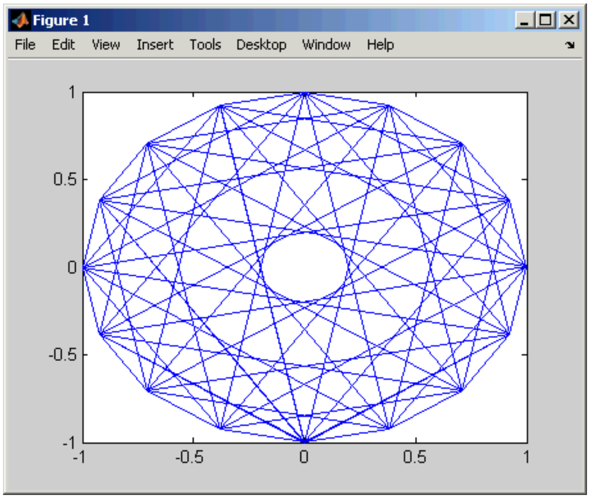
`y = dpskmod(x,M,phaserot)` specifies the phase rotation of the modulation in radians. In this case, the total phase shift per symbol is the sum of `phaserot` and the phase generated by the differential modulation.

`y = dpskmod(x,M,phaserot,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray-coded ordering.

Examples

The example below plots the output of the `dpskmod` function. The image shows the possible transitions from each symbol in the DPSK signal constellation to the next symbol.

```
s = RandStream.create('mt19937ar', 'seed',131);
prevStream = RandStream.setGlobalStream(s); % seed for repeatability
M = 4; % Use DQPSK in this example, so M is 4.
x = randi([0 M-1],500,1); % Random data
y = dpskmod(x,M,pi/8); % Modulate using a nonzero initial phase.
plot(y) % Plot all points, using lines to connect them.
```



For another example that uses this function, see "Example: Curve Fitting for an Error Rate Plot".

See Also

`dpskdemod` | `pskmod` | `pskdemod`

How To

- "Digital Modulation"

Purpose

Low-density parity-check codes from DVB-S.2 standard

Syntax

```
H = dvbs2ldpc(r)
```

Description

`H = dvbs2ldpc(r)` returns the parity-check matrix of the LDPC code with code rate `r` from the DVB-S.2 standard. `H` is a sparse logical matrix.

Possible values for `r` are 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 8/9, and 9/10. The block length of the code is 64800.

The default parity-check matrix (32400-by-64800) corresponds to an irregular LDPC code with the structure shown in the following table.

Row	Number of 1s Per Row
1	6
2 to 32400	7

Column	Number of 1s Per Column
1 to 12960	8
12961 to 32400	3

Columns 32401 to 64800 form a lower triangular matrix. Only the elements on its main diagonal and the subdiagonal immediately below are 1s. This LDPC code is used in conjunction with a BCH code in the Digital Video Broadcasting standard DVB-S.2 to achieve a packet error rate below 10^{-7} at about 0.7 dB to 1 dB from the Shannon limit.

Examples

```
H = dvbs2ldpc(3/5);
spy(H);   % Visualize the location of nonzero elements in H.
henc = comm.LDPCEncoder(H);
hdec = comm.LDPCDecoder(H);
```

How To

- `spy`

encode

Purpose

Block encoder

Syntax

```
code = encode(msg,n,k,'linear/fmt',genmat)
code = encode(msg,n,k,'cyclic/fmt',genpoly)
code = encode(msg,n,k,'hamming/fmt',prim_poly)
code = encode(msg,n,k)
[code,added] = encode(...)
```

Optional Inputs

Input	Default Value
fmt	binary
genpoly	cyclpoly(n,k)
prim_poly	gfprimdf(n-k)

Description

For All Syntaxes

The encode function encodes messages using one of the following error-correction coding methods:

- Linear block
- Cyclic
- Hamming

For all of these methods, the codeword length is n and the message length is k.

msg, which represents the messages, can have one of several formats. The table below shows which formats are allowed for msg, how the argument *fmt* should reflect the format of msg, and how the format of the output code depends on these choices. The examples in the table are for k = 4. If *fmt* is not specified as input, its default value is binary.

Note If `2^n` or `2^k` is large, use the default `binary` format instead of the `decimal` format. This is because the function uses a binary format internally, while the roundoff error associated with converting many bits to large decimal numbers and back might be substantial.

Information Formats

Dimension of `msg`	Value of *fmt* Argument	Dimension of `code`
Binary column or row vector	`binary`	Binary column or row vector
Example: `msg = [0 1 1 0, 0 1 0 1, 1 0 0 1].'`		
Binary matrix with `k` columns	`binary`	Binary matrix with `n` columns
Example: `msg = [0 1 1 0; 0 1 0 1; 1 0 0 1]`		
Column or row vector of integers in the range [0, `2^k-1`]	`decimal`	Column or row vector of integers in the range [0, `2^n-1`]
Example: `msg = [6, 10, 9].'`		

For Specific Syntaxes

`code = encode(msg,n,k,'linear/`*`fmt`*`',genmat)` encodes `msg` using `genmat` as the generator matrix for the linear block encoding method. `genmat`, a `k`-by-`n` matrix, is required as input.

`code = encode(msg,n,k,'cyclic/`*`fmt`*`',genpoly)` encodes `msg` and creates a systematic cyclic code. `genpoly` is a row vector that gives the coefficients, in order of ascending powers, of the binary generator polynomial. The default value of `genpoly` is `cyclpoly(n,k)`. By definition, the generator polynomial for an [n,k] cyclic code must have degree n-k and must divide x^n-1.

`code = encode(msg,n,k,'hamming/fmt',prim_poly)` encodes `msg` using the Hamming encoding method. For this syntax, `n` must have the form 2^m-1 for some integer m greater than or equal to 3, and `k` must equal `n`-m. `prim_poly` is a row vector that gives the binary coefficients, in order of ascending powers, of the primitive polynomial for GF(2^m) that is used in the encoding process. The default value of `prim_poly` is the default primitive polynomial `gfprimdf(m)`.

`code = encode(msg,n,k)` is the same as `code = encode(msg,n,k,'hamming/binary')`.

`[code,added] = encode(...)` returns the additional variable `added`. `added` is the number of zeros that were placed at the end of the message matrix before encoding in order for the matrix to have the appropriate shape. “Appropriate” depends on `n`, `k`, the shape of `msg`, and the encoding method.

Examples

The example below illustrates the three different information formats (binary vector, binary matrix, and decimal vector) for Hamming code. The three messages have identical content in different formats; as a result, the three codes that `encode` creates have identical content in correspondingly different formats.

```
m = 4; n = 2^m-1; % Codeword length = 15
k = 11; % Message length

% Create 100 messages, k bits each.
msg1 = randi([0,1],100*k,1); % As a column vector
msg2 = vec2mat(msg1,k); % As a k-column matrix
msg3 = bi2de(msg2)'; % As a row vector of decimal integers

% Create 100 codewords, n bits each.
code1 = encode(msg1,n,k,'hamming/binary');
code2 = encode(msg2,n,k,'hamming/binary');
code3 = encode(msg3,n,k,'hamming/decimal');
if ( vec2mat(code1,n)==code2 & de2bi(code3',n)==code2 )
   disp('All three formats produced the same content.')
end
```

The output is

```
All three formats produced the same content.
```

The next example creates a cyclic code, adds noise, and then decodes the noisy code. It uses the `decode` function.

```
n = 3; k = 2; % A (3,2) cyclic code
msg = randi([0,1],100,k); % 100 messages, k bits each
code = encode(msg,n,k,'cyclic/binary');
% Add noise.
noisycode = rem(code + randerr(100,n,[0 1;.7 .3]), 2);
newmsg = decode(noisycode,n,k,'cyclic'); % Try to decode.
% Compute error rate for decoding the noisy code.
[number,ratio] = biterr(newmsg,msg);
disp(['The bit error rate is ',num2str(ratio)])
```

The output is below. Your error rate results might vary because the noise is random.

```
The bit error rate is 0.08
```

The next example encodes the same message using Hamming and cyclic methods. This example also creates Hamming code with the `'linear'` option of the `encode` command. It then decodes each code and recovers the original message.

```
n = 7; % Codeword length
k = 4; % Message length
m = log2(n+1); % Express n as 2^m-1.
msg = randi([0,2^k-1],100,1); % Column of decimal integers

% Create various codes.
codehamming = encode(msg,n,k,'hamming/decimal');
[parmat,genmat] = hammgen(m);
codehamming2 = encode(msg,n,k,'linear/decimal',genmat);
if codehamming==codehamming2
   disp('The ''linear'' method can create Hamming code.')
```

```
end
codecyclic = encode(msg,n,k,'cyclic/decimal');

% Decode to recover the original message.
decodedhamming = decode(codehamming,n,k,'hamming/decimal');
decodedcyclic = decode(codecyclic,n,k,'cyclic/decimal');
if (decodedhamming==msg & decodedcyclic==msg)
   disp('All decoding worked flawlessly in this noiseless world.')
end
```

The output is

```
The 'linear' method can create Hamming code.
All decoding worked flawlessly in this noiseless world.
```

Algorithms

Depending on the encoding method, encode relies on such lower-level functions as hammgen and cyclgen.

See Also

decode | cyclpoly | cyclgen | hammgen

How To

- “Block Codes”

Purpose

Equalize signal using equalizer object

Syntax

```
y = equalize(eqobj,x)
y = equalize(eqobj,x,trainsig)
[y,yd] = equalize(...)
[y,yd,e] = equalize(...)
```

Description

`y = equalize(eqobj,x)` processes the baseband signal vector `x` with equalizer object `eqobj` and returns the equalized signal vector `y`. At the end of the process, `eqobj` contains updated state information such as equalizer weight values and input buffer values. To construct `eqobj`, use the `lineareq` or `dfe` function, as described in “Adaptive Algorithms”. The `equalize` function assumes that the signal `x` is sampled at `nsamp` samples per symbol, where `nsamp` is the value of the `nSampPerSym` property of `eqobj`. For adaptive algorithms other than CMA, the equalizer adapts in decision-directed mode using a detector specified by the `SigConst` property of `eqobj`. The delay of the equalizer is `(eqobj.RefTap-1)/eqobj.nSampPerSym`, as described in “Delays from Equalization”.

Note that `(eqobj.RefTap-1)` must be an integer multiple of `nSampPerSym`. For a fractionally-spaced equalizer, the taps are spaced at fractions of a symbol period. The reference tap pertains to training symbols, and thus, must coincide with a whole number of symbols (i.e., an integer number of samples per symbol). `eqobj.RefTap=1` corresponds to the first symbol, `eqobj.RefTap=nSampPerSym+1` to the second, and so on. Therefore `(eqobj.RefTap-1)` must be an integer multiple of `nSampPerSym`.

If `eqobj.ResetBeforeFiltering` is `0`, `equalize` uses the existing state information in `eqobj` when starting the equalization operation. As a result, `equalize(eqobj,[x1 x2])` is equivalent to `[equalize(eqobj,x1) equalize(eqobj,x2)]`. To reset `eqobj` manually, apply the `reset` function to `eqobj`.

If `eqobj.ResetBeforeFiltering` is `1`, `equalize` resets `eqobj` before starting the equalization operation, overwriting any previous state information in `eqobj`.

`y = equalize(eqobj,x,trainsig)` initially uses a training sequence to adapt the equalizer. After processing the training sequence, the equalizer adapts in decision-directed mode. The vector length of `trainsig` must be less than or equal to `length(x)-(eqobj.RefTap-1)/eqobj.nSampPerSym`.

`[y,yd] = equalize(...)` returns the vector `yd` of detected data symbols.

`[y,yd,e] = equalize(...)` returns the result of the error calculation described in "Error Calculation". For adaptive algorithms other than CMA, `e` is the vector of errors between `y` and the reference signal, where the reference signal consists of the training sequence or detected symbols.

Examples

For examples that use this function, see "Equalize Using a Training Sequence in MATLAB", "Example: Equalizing Multiple Times, Varying the Mode", and "Example: Adaptive Equalization Within a Loop".

See Also

`lms` | `signlms` | `normlms` | `varlms` | `rls` | `cma` | `lineareq` | `dfe`

How To

- "Equalization"

Purpose

Generate eye diagram

Syntax

```
eyediagram(x,n)
eyediagram(x,n,period)
eyediagram(x,n,period,offset)
eyediagram(x,n,period,offset,plotstring)
eyediagram(x,n,period,offset,plotstring,h)
h = eyediagram(...)
```

Description

`eyediagram(x,n)` creates an eye diagram for the signal `x`, plotting `n` samples in each trace. `n` must be an integer greater than 1. The labels on the horizontal axis of the diagram range between -1/2 and 1/2. The function assumes that the first value of the signal, and every `n`th value thereafter, occur at integer times. The interpretation of `x` and the number of plots depend on the shape and complexity of `x`:

- If `x` is a real two-column matrix, `eyediagram` interprets the first column as in-phase components and the second column as quadrature components. The two components appear in different subplots of a single figure window.
- If `x` is a complex vector, `eyediagram` interprets the real part as in-phase components and the imaginary part as quadrature components. The two components appear in different subplots of a single figure window.
- If x is a real vector, `eyediagram` interprets it as a real signal. The figure window contains a single plot.

`eyediagram(x,n,period)` is the same as the syntax above, except that the labels on the horizontal axis range between `-period/2` and `period/2`.

`eyediagram(x,n,period,offset)` is the same as the syntax above, except that the function assumes that the (`offset+1`)st value of the signal, and every `n`th value thereafter, occur at times that are integer multiples of `period`. The variable `offset` must be a nonnegative integer between 0 and `n-1`.

`eyediagram(x,n,period,offset,plotstring)` is the same as the syntax above, except that `plotstring` determines the plotting symbol, line type, and color for the plot. `plotstring` is a string whose format and meaning are the same as in the `plot` function. The default string is `'b-'`, which produces a blue solid line.

`eyediagram(x,n,period,offset,plotstring,h)` is the same as the syntax above, except that the eye diagram is in the figure whose handle is `h`, rather than in a new figure. `h` must be a handle to a figure that `eyediagram` previously generated.

Note You cannot use `hold on` to plot multiple signals in the same figure.

`h = eyediagram(...)` is the same as the earlier syntaxes, except that `h` is the handle to the figure that contains the eye diagram.

Examples

For an online demonstration, type `showdemo scattereyedemo`.

See Also

`scatterplot | plot`

How To

- "Eye Diagram Analysis"

Purpose

Launch eye diagram scope for eye diagram object H

Syntax

```
eyescope
eyescope(h)
```

Description

Use EyeScope to examine the data in an eye diagram object. EyeScope shows both the eye diagram plot and measurement results in a unified, graphical environment, providing a very efficient means for viewing eye diagram data. There are two ways to call EyeScope:

- `eyescope` calls an empty scope
- `eyescope(h)` call EyeScope, displaying object *h*

Note You can call EyeScope with an eye diagram object as the input argument. EyeScope uses the `inputname` function to resolve the caller's work space name for the argument. If the `inputname` function cannot resolve the caller's work space name, then EyeScope uses a default name. To learn about the cases when EyeScope can not determine the work space name, type `help inputname` at the MATLAB command line.

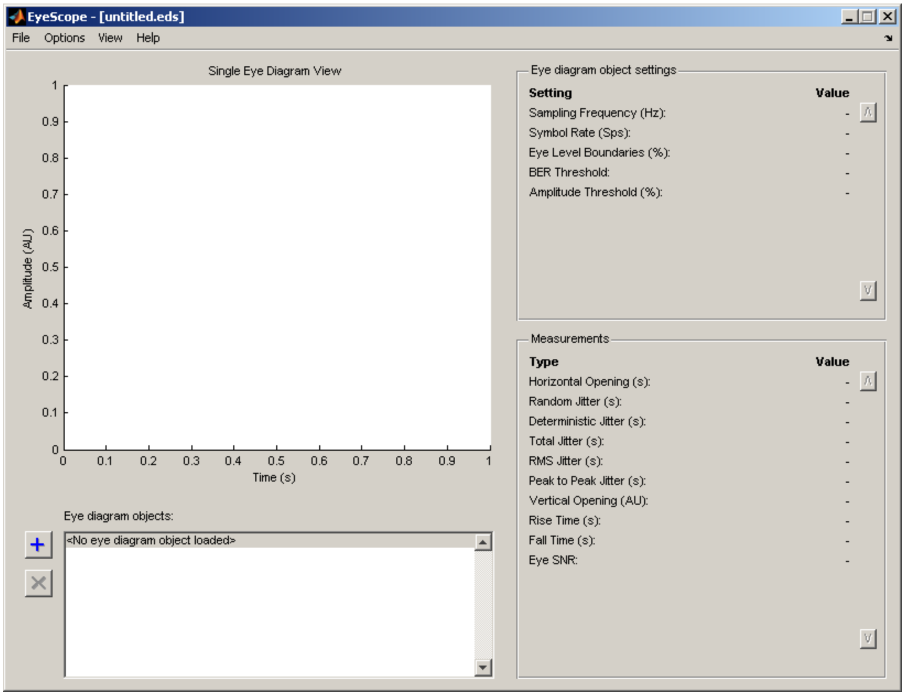
For more information, see "Eye Diagram Analysis".

Starting EyeScope

To start EyeScope from the MATLAB command line, type:

```
eyescope
```

The following figure shows an EyeScope that does not have an eye diagram object loaded in its memory.



Alternatively, you can start EyeScope so it displays an eye diagram object. To start EyeScope so it displays an eye diagram object, type the following at the MATLAB command line:

```
eyescope(h)
```

Note *h* is a handle to an eye diagram object in the workspace.

The EyeScope Environment

- "EyeScope Menu Bar" on page 1-245
- "Eye Diagram Object Plot and Plot Controls" on page 1-245
- "Eye Diagram Object Settings Panel" on page 1-247

- "Measurements" on page 1-248

EyeScope Menu Bar

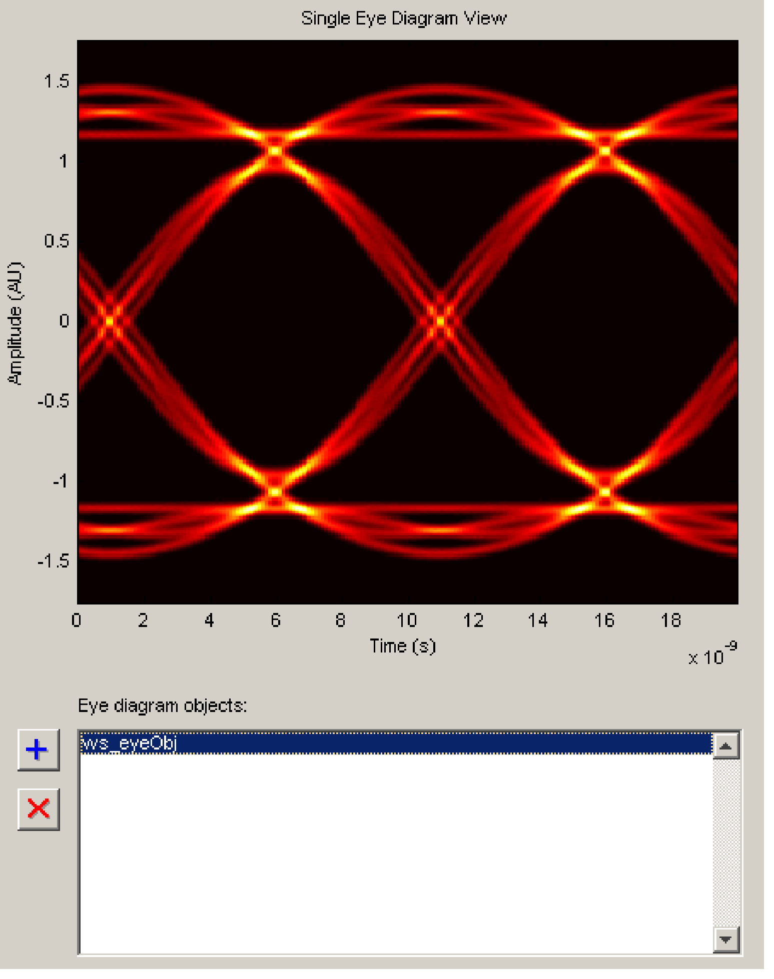
EyeScope Menu Bar

The EyeScope menu bar is comprised of four menus: File, Options, View, and Help.

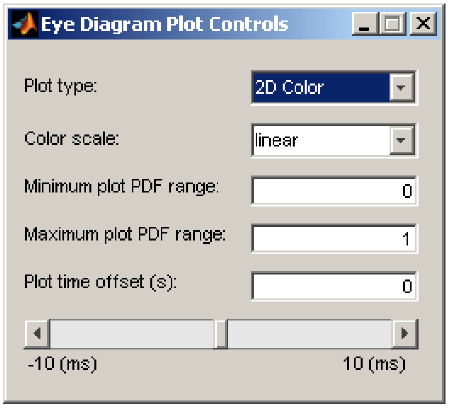
- Use the **File** menu to control the session management functions, import an eye diagram object into EyeScope, and export an eye diagram plot.
- Use the **Options** menu to setup the eye diagram scope by selecting which eye diagram settings and measurements EyeScope displays.
- Use the **View** menu to toggle between Single eye diagram view or Compare measurement results view, and to add or modify a legend for the eye diagram plot.
- The **Help** menu is used to access help pertaining to the eye diagram object and EyeScope.

Eye Diagram Object Plot and Plot Controls

The Eye diagram object plot is the region of the GUI where the eye diagram plot appears.



Eye diagram plot controls are user-configurable settings that specify plot type, color scale, minimum and maximum plot PDF range, and plot time offset for the eye diagram being analyzed. To access the EyeScope plot controls **Options > Eye Diagram Plot Controls**



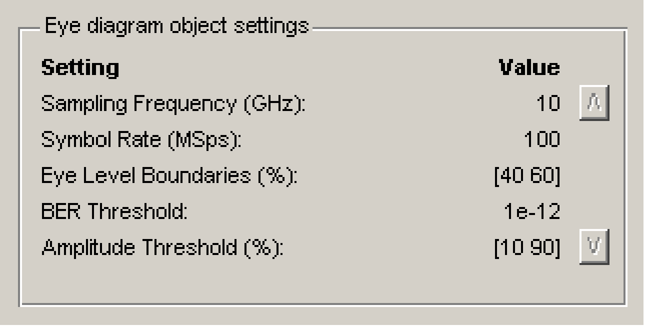
Note The value for the **Plot time offset** parameter can either be entered directly into the text box or set using the slide bar control.

For more information pertaining to the eye diagram properties, refer to the `commscope.eyediagram` reference page.

Eye Diagram Object Settings Panel

The eye diagram object settings panel displays the eye diagram object settings. The default EyeScope configuration displays the following eye diagram object settings:

- Sampling frequency
- Symbol rate
- Eye level boundaries
- BER threshold
- Amplitude threshold

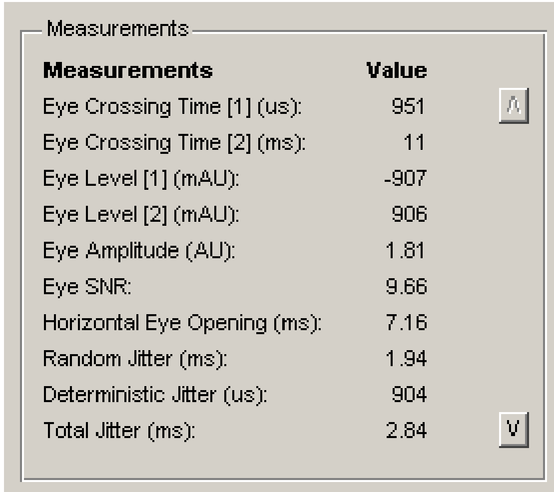


To specify which eye diagram object settings display in EyeScope, refer to "Selecting Which Eye Diagram Object Settings To Display" on page 1-255. If you select additional eye diagram object settings to display in EyeScope, use the scroll buttons to view all of the settings.

Measurements

The Measurements panel displays the eye diagram measurement settings. The default EyeScope configuration displays the following eye diagram object measurements:

- Horizontal Eye Opening
- Random Jitter
- Deterministic Jitter
- Total Jitter
- RMS Jitter
- Peak to Peak Jitter
- Vertical Opening
- Rise Time
- Fall Time
- Eye SNR



To select which eye diagram measurements EyeScope displays, refer to "Selecting Which Eye Diagram Measurements To Display" on page 1-256. If you select additional eye diagram object measurements to display in EyeScope, use the scroll buttons to view all of the settings.

Using EyeScope

- "Starting EyeScope with an Argument" on page 1-250
- "Starting a new Session" on page 1-250
- "Opening a Session" on page 1-250
- "Saving a Session" on page 1-251
- "Importing an Eye Diagram Object" on page 1-252
- "Printing to a Figure" on page 1-254
- "Selecting Which Eye Diagram Object Settings To Display" on page 1-255
- "Selecting Which Eye Diagram Measurements To Display" on page 1-256

Starting EyeScope with an Argument

You can start EyeScope so it is displaying an eye diagram object. To start EyeScope so it is displaying an eye diagram object, type the following at the MATLAB command line:

```
eyescope(h)
```

Note *h* is a handle to an eye diagram object presently in the workspace.

Starting a new Session

Starting a new session purges EyeScope memory, returning EyeScope to an empty plot display. If changes have been made to an open session and you start a new session, you will be prompted to save the open session.

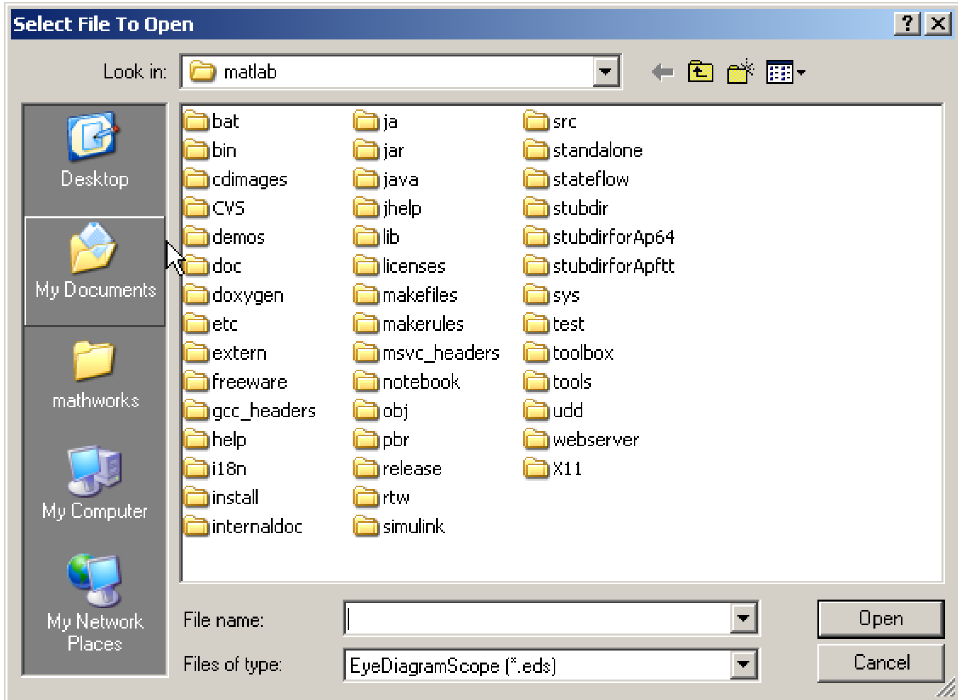
Opening a Session

To open session, choose the file name and location of the session file. The file extensions for a session file is .eds, which stands for eye diagram scope. If changes have been made to a session that is presently open and you try to open up a new session, you will be prompted to save the session that is presently open before the new session can start.

To open a session:

1. Click **File > Open Session**.

   The Select File To Open Window appears.



2

Navigate to the EyeScope session file you want, and click **Open**.

Saving a Session

The Save Session selection saves the current session, updating the session file. A session file includes the eye diagram object, eyescope options, and plot control selections.

If you attempt to save a session that you have not previously saved, EyeScope will prompt you for a file name and location. Otherwise, the session is saved to the previously selected file.

To save a session, follow these steps:

1

Click **File > Save Session**.

2

Navigate to the folder where you want to save the EyeScope session file and click **Save**.

Importing an Eye Diagram Object

The **Import** menu selection imports an eye diagram object from either the workspace or a MAT-file to EyeScope. The imported variable name will be reconstructed to reflect the origin of the eye diagram object, as follows:

- If an object is imported from the workspace, the variable name will be *ws_object name*, where *object name* is the name of the original variable.
- If the object is imported from a MATLAB file, then the file name (without the path) precedes the object name.

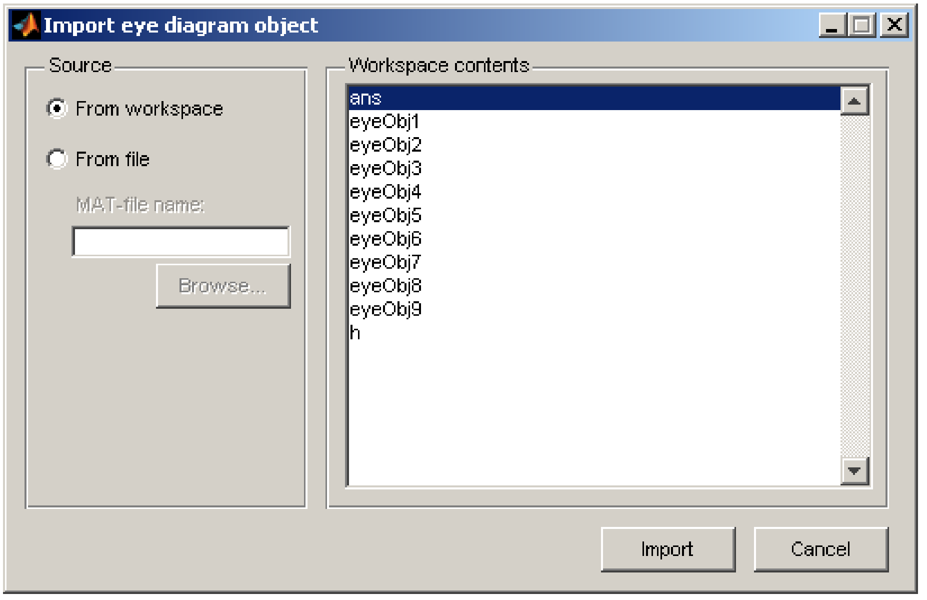
Importing an object creates a copy of the object, using the naming convention previously described. EyeScope displays the object's contents as configured when the object was imported. EyeScope does not track any object changes made in the workspace (or to the MATLAB file) from which the object was imported.

To import an eye diagram object:

1

Click **File > Import Eye Diagram Object**

The Import eye diagram object window appears.



The contents panel of the of the Import eye diagram object window displays all eye diagram objects available in the source location.

2

From the Import eye diagram object window, select the source for the object being imported.

- Select **From workspace** to import an eye diagram object directly from the workspace.
- Select **From File** to choose an eye diagram object file that was previously saved and click **Browse** to select the file to be loaded.

3

Click **Import**.

Printing to a Figure

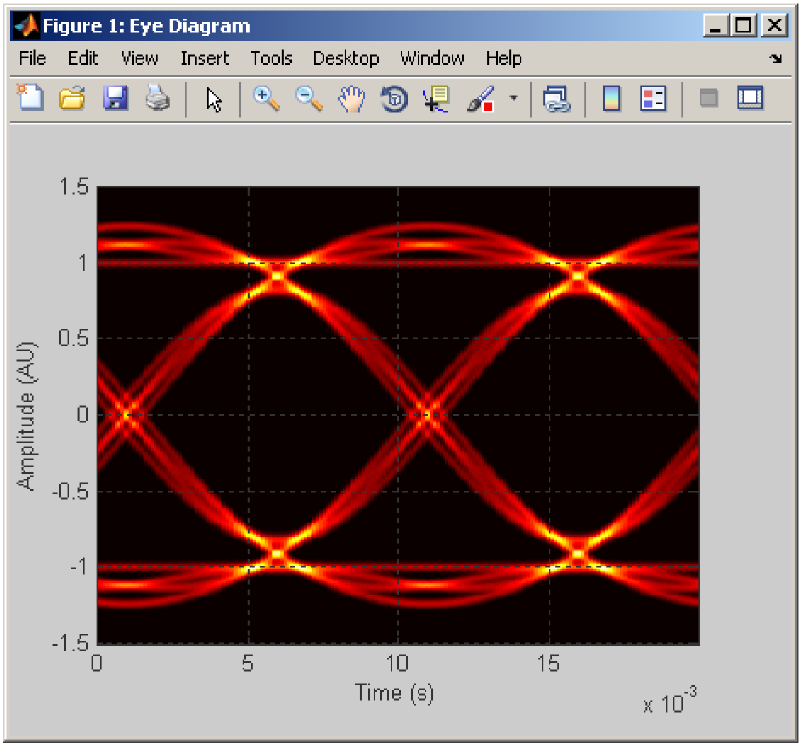
EyeScope allows you to print an eye diagram plot to a separate MATLAB figure window. From the MATLAB figure window, along with other tasks, you can print, zoom, or edit the plot.

To export an eye diagram figure:

1

Click **File > Print to Figure**

The MATLAB figure window, containing the exported image, appears.



Selecting Which Eye Diagram Object Settings To Display

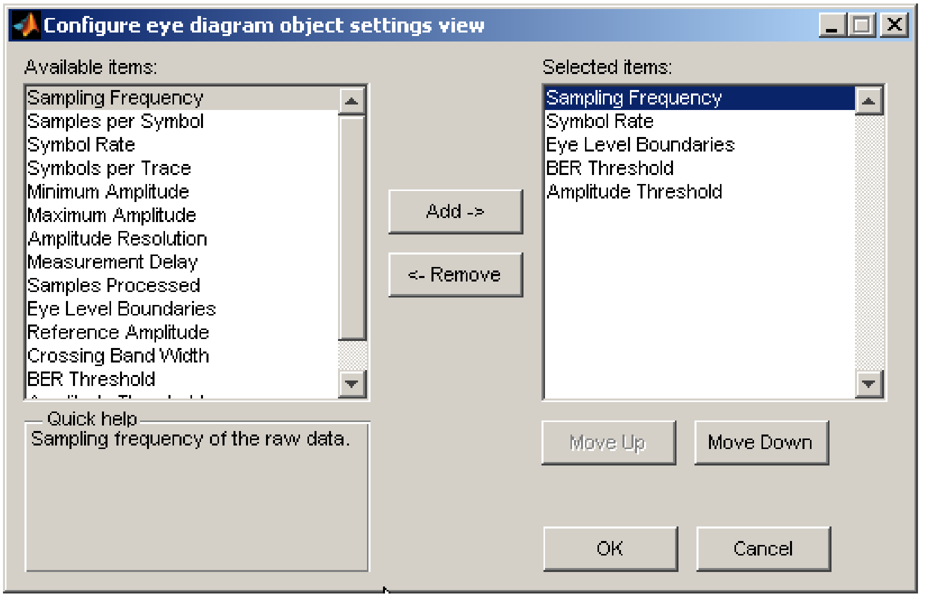
The **Eye Diagram Object Settings View** allows you to select which object settings display in the eye diagram object settings panel. You make your selections in the Configure eye diagram object settings view window, where a shuttle control allows you to add, remove, or reorder the settings you are displaying.

To add an eye diagram object setting:

1

Click **Options > Eye Diagram Object Settings View**

The Configure eye diagram object settings view window appears.



2

Locate any items to be added in the list of **Available items**, and left-click to select.

Note To select multiple items, you can either press and hold the <Shift> key and left-click or press and hold the <Ctrl> key and left-click.

When you select an item, the **Quick help** panel displays information about the item. If you select multiple items, **Quick help** displays information pertaining to the last item you select.

3

Click **Add**.

Note Using the **Move Up** or**Move Down** buttons, you can change the order in which the eye diagrams settings you select appear.

4

Click **OK** .

Selecting Which Eye Diagram Measurements To Display

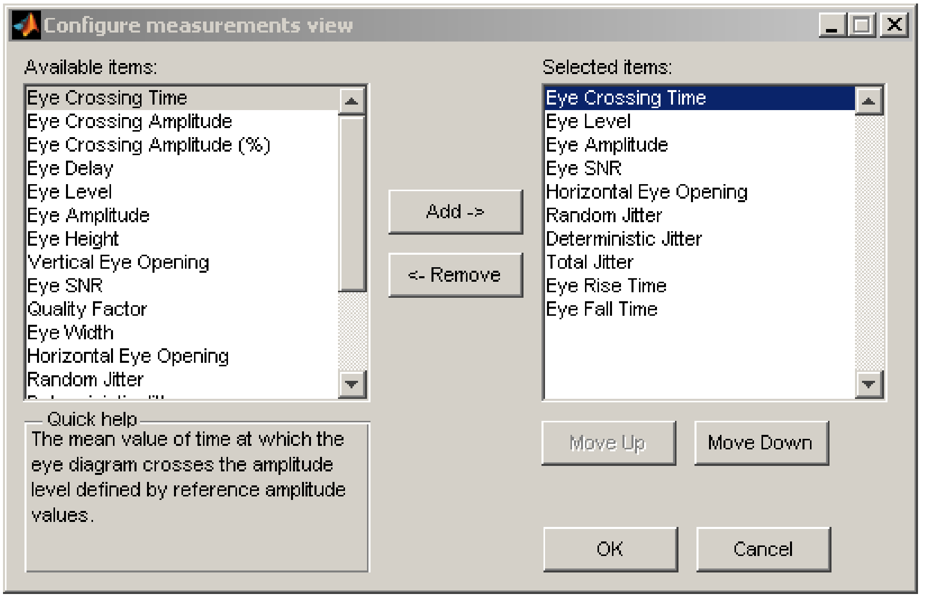
You can modify the contents of the measurement panel by selecting which eye diagram measurements display in the eye diagram object settings panel. You make your selections in the Configure measurements view window, where a shuttle control allows you to add, remove, or reorder the settings you are including.

Adding An Eye Diagram Measurement Setting

1

Click **Options > Measurements View**

The Configure measurements window appears.



2

Locate any items to be added in the list of **Available items**, and left-click to select.

Note To select multiple items, you can either press and hold the <Shift> key and left-click or press and hold the <Ctrl> key and left-click.

When you select an item, the **Quick help** panel displays information about the item. If you select multiple items, **Quick help** displays information pertaining to the last item you select.

3

Click **Add**.

Note Using the **Move Up** or **Move Down** buttons, you can change the order in which the eye diagrams settings you select appear.

4

Click **OK** .

Purpose

Discrete Fourier transform

Syntax

```
fft(x)
```

Description

`fft(x)` is the discrete Fourier transform (DFT) of the Galois vector `x`. If `x` is in the Galois field GF(2^m), the length of `x` must be 2^m-1.

Examples

```
m = 4;
n = 2^m-1;
x = gf(randi([0 2^m-1],n,1),m); % Random vector
y = fft(x); % Transform of x
z = ifft(y); % Inverse transform of y
ck = isequal(z,x) % Check that ifft(fft(x)) recovers x.
```

The output is

```
ck =

     1
```

Limitations

The Galois field over which this function works must have 256 or fewer elements. In other words, `x` must be in the Galois field GF(2^m), where m is an integer between 1 and 8.

Algorithms

If `x` is a column vector, `fft` applies `dftmtx` to the primitive element of the Galois field and multiplies the resulting matrix by `x`.

See Also

`ifft` | `dftmtx`

How To

- "Signal Processing Operations in Galois Fields"

filter (channel)

Purpose

Filter signal with channel object

Syntax

```
y = filter(chan,x)
```

Description

`y = filter(chan,x)` processes the baseband signal vector `x` with the channel object `chan`. The result is the signal vector `y`. The final state of the channel is stored in `chan`. You can construct `chan` using either `rayleighchan` or `riciananchan`. The `filter` function assumes `x` is sampled at frequency `1/ts`, where `ts` equals the `InputSamplePeriod` property of `chan`.

If `chan.ResetBeforeFiltering` is `0`, `filter` uses the existing state information in `chan` when starting the filtering operation. As a result, `filter(chan,[x1 x2])` is equivalent to `[filter(chan,x1) filter(chan,x2)]`. To reset `chan` manually, apply the `reset` function to `chan`.

If `chan.ResetBeforeFiltering` is `1`, `filter` resets `chan` before starting the filtering operation, overwriting any previous state information in `chan`.

Examples

Examples using this function are in “Use Fading Channels”.

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

See Also

`rayleighchan` | `ricianchan` | `reset`

How To

- “Fading Channels”

Purpose

1-D digital filter over Galois field

Syntax

```
y = filter(b,a,x)
[y,zf] = filter(b,a,x)
```

Description

`y = filter(b,a,x)` filters the data in the vector `x` with the filter described by numerator coefficient vector `b` and denominator coefficient vector `a`. The vectors `b`, `a`, and `x` must be Galois vectors in the same field. If `a(1)` is not equal to `1`, `filter` normalizes the filter coefficients by `a(1)`. As a result, `a(1)` must be nonzero.

The filter is a "Direct Form II Transposed" implementation of the standard difference equation below.

```
a(1)*y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb) ...
                      - a(2)*y(n-1) - ... - a(na+1)*y(n-na)
```

`[y,zf] = filter(b,a,x)` returns the final conditions of the filter delays in the Galois vector `zf`. The length of the vector `zf` is `max(size(a),size(b))-1`.

Examples

An example is in "Huffman Coding".

Purpose

Estimate delay(s) between signals

Syntax

```
D = finddelay(X,Y)
D = finddelay(...,MAXLAG)
```

Description

`D = finddelay(X,Y)`, where `X` and `Y` are row or column vectors, returns an estimate of the delay `D` between `X` and `Y`, where `X` serves as the reference vector. If `Y` is delayed with respect to `X`, then `D` is positive. If `Y` is advanced with respect to `X`, then `D` is negative. Delays in `X` and `Y` can be introduced by pre-pending zeros.

`X` and `Y` need not be exact delayed copies of each other, as `finddelay(X,Y)` returns an estimate of the delay via cross-correlation. However this estimated delay has a useful meaning only if there is sufficient correlation between delayed versions of `X` and `Y`. Also, if several delays are possible, as in the case of periodic signals, the delay with the smallest absolute value is returned. In the case that both a positive and a negative delay with the same absolute value are possible, the positive delay is returned.

`D = finddelay(X,Y)`, where `X` is a matrix of size MX-by-NX (MX>1 and NX>1) and `Y` is a matrix of size MY-by-NY (MY>1 and NY>1), returns a row vector `D` of estimated delays between each column of `X` and the corresponding column of `Y`. With this usage the number of columns of `X` must be equal to the number of columns of `Y` (i.e., NX=NY).

`D = finddelay(...,MAXLAG)`, uses `MAXLAG` as the maximum correlation window size used to find the estimated delay(s) between `X` and `Y`. The usage of `MAXLAG` is detailed in the table below.

By default, `MAXLAG` is equal to MAX(LX, LY)-1 for two vector inputs (where LX and LY are the lengths of `X` and `Y`, respectively), MAX(MX, MY)-1 for two matrix inputs, and MAX(LX, MY)-1 or MAX(MX, LY)-1 for one vector input and one matrix input. If `MAXLAG` is input as `[]`, it is replaced by the default value. If any element of `MAXLAG` is negative, it is replaced by its absolute value. If any element of `MAXLAG` is not integer-valued, or is complex, `Inf`, or `NaN`, then `finddelay` returns an error.

The calculation of the vector of estimated delays, `D`, depends on `X`, `Y`, and `MAXLAG` as shown in the following table.

`MAXLAG`	`X`	`Y`	`D` is calculated by...
Integer-valued scalar	Row or column vector or matrix	Row or column vector or matrix	Cross-correlating the columns of `X` and `Y` over a range of lags `-MAXLAG:MAXLAG`.
Integer-valued row or column vector	Row or column vector of length $LX \geq 1$	Matrix of size *MY*-by-*NY* (*MY*>1, *NY*>1)	Cross-correlating `X` and column *j* of `Y` over a range of lags `-MAXLAG(`*j*`):MAXLAG(`*j*`)`, for *j*=1:*NY*.
Integer-valued row or column vector	Matrix of size *MX*-by-*NX* (*MX*>1, *NX*>1)	Row or column vector of length $LY \geq 1$	Cross-correlating column *j* of `X` and `Y` over a range of lags `-MAXLAG(`*j*`):MAXLAG(`*j*`)`, for *j*=1:*NX*.
Integer-valued row or column vector	Matrix of size *MX*-by-*NX* (*MX*>1, *NX*>1)	Matrix of size *MY*-by-*NY* (*MY*>1, *NY*=*NX*>1)	Cross-correlating column *j* of `X` and column *j* of `Y` over a range of lags `-MAXLAG(`*j*`):MAXLAG(`*j*`)`, for *j*=1:*NY*.

Treating `X` as Multiple Channels

If you wish to treat a row vector `X` of length *LX* as comprising one sample from *LX* different channels, you need to append one or more rows of zeros to `X` so that it appears as a matrix. Then each column of `X` will be considered a channel.

For example, `X = [1 1 1 1]` is considered a single channel comprising four samples. To treat it as four different channels, each channel comprising one sample, define a new matrix *Xm*:

```
Xm = [1 1 1 1;
      0 0 0 0];
```

Each column of *Xm* corresponds to a single channel, each one containing the samples `1` and `0`.

Algorithms

The `finddelay` function uses the `xcorr` function of Signal Processing Toolbox to determine the cross-correlation between each pair of signals at all possible lags specified by the user. The normalized cross-correlation between each pair of signals is then calculated. The estimated delay is given by the negative of the lag for which the normalized cross-correlation has the largest absolute value.

If more than one lag leads to the largest absolute value of the cross-correlation, such as in the case of periodic signals, the delay is chosen as the negative of the smallest (in absolute value) of such lags.

Pairs of signals need not be exact delayed copies of each other. However, the estimated delay has a useful meaning only if there is sufficient correlation between at least one pair of the delayed signals.

Examples

X and Y Are Vectors, and MAXLAG Is Not Specified

The following shows `Y` being delayed with respect to `X` by two samples.

```
X = [1 2 3];
Y = [0 0 1 2 3];
D = finddelay(X,Y)
```

The result is `D = 2`.

Here is a case of `Y` advanced with respect to `X` by three samples.

```
X = [0 0 0 1 2 3 0 0]';
Y = [1 2 3 0]';
D = finddelay(X,Y)
```

The result is `D = -3`.

The following illustrates a case where `Y` is aligned with `X` but is noisy.

```
X = [0 0 1 2 3 0];
Y = [0.02 0.12 1.08 2.21 2.95 -0.09];
D = finddelay(X,Y)
```

The result is `D = 0`.

If Y is a periodic version of X, the smallest possible delay is returned.

```
X = [0 1 2 3];
Y = [1 2 3 0 0 0 0 1 2 3 0 0];
D = finddelay(X,Y)
```

The result is `D = -1`.

X is a Vector, Y a Matrix, and MAXLAG Is a Scalar

`MAXLAG` is specified as a scalar (same maximum window sizes).

```
X = [0 1 2];
Y = [0 1 0 0;
     1 2 0 0;
     2 0 1 0;
     0 0 2 1];
MAXLAG = 3;
D = finddelay(X,Y,MAXLAG)
```

The result is `D = [0 -1 1 1]`.

X and Y Are Matrices, and MAXLAG Is Not Specified

```
X = [0 1 0 0;
     1 2 0 0;
     2 0 1 0;
     1 0 2 1;
     0 0 0 2];
Y = [0 0 1 0;
     1 1 2 0;
     2 2 0 1;
     1 0 0 2;
     0 0 0 0];
D = finddelay(X,Y)
```

The result is `D = [0 1 -2 -1]`.

X and Y Are Matrices, and MAXLAG Is Specified

```
X = [0 1 0 0;
     1 2 0 0;
     2 0 1 0;
     1 0 2 1;
     0 0 0 2];
Y = [0 0 1 0;
     1 1 2 0;
     2 2 0 1;
     1 0 0 2;
     0 0 0 0];
MAXLAG = [10 10 20 20];
D = finddelay(X,Y,MAXLAG)
```

The result is `D = [0 1 -2 -1]`.

See Also

`alignsignals` | `xcorr`

Purpose

Frequency demodulation

Syntax

```
z = fmdemod(y,Fc,Fs,freqdev)
z = fmdemod(y,Fc,Fs,freqdev,ini_phase)
```

Description

`z = fmdemod(y,Fc,Fs,freqdev)` demodulates the modulating signal `z` from the carrier signal using frequency demodulation. The carrier signal has frequency `Fc` (Hz) and sampling rate `Fs` (Hz), where `Fs` must be at least `2*Fc`. The `freqdev` argument is the frequency deviation (Hz) of the modulated signal `y`.

`z = fmdemod(y,Fc,Fs,freqdev,ini_phase)` specifies the initial phase of the modulated signal, in radians.

Examples

An example using `fmdemod` is on the reference page for `fmmod`.

See Also

`fmmod` | `pmmod` | `pmdemod`

How To

- "Digital Modulation"

fmmod

Purpose

Frequency modulation

Syntax

```
y = fmmod(x,Fc,Fs,freqdev)
y = fmmod(x,Fc,Fs,freqdev,ini_phase)
```

Description

`y = fmmod(x,Fc,Fs,freqdev)` modulates the message signal `x` using frequency modulation. The carrier signal has frequency `Fc` (Hz) and sampling rate `Fs` (Hz), where `Fs` must be at least `2*Fc`. The `freqdev` argument is the frequency deviation constant (Hz) of the modulated signal.

`y = fmmod(x,Fc,Fs,freqdev,ini_phase)` specifies the initial phase of the modulated signal, in radians.

Examples

The code below modulates a multichannel signal using `fmmod` and demodulates it using `fmdemod`.

```
Fs = 8000; % Sampling rate of signal
Fc = 3000; % Carrier frequency
t = [0:Fs-1]'/Fs; % Sampling times
s1 = sin(2*pi*300*t)+2*sin(2*pi*600*t); % Channel 1
s2 = sin(2*pi*150*t)+2*sin(2*pi*900*t); % Channel 2
x = [s1,s2]; % Two-channel signal
dev = 50; % Frequency deviation in modulated signal
y = fmmod(x,Fc,Fs,dev); % Modulate both channels.
z = fmdemod(y,Fc,Fs,dev); % Demodulate both channels.
plot(z);
```

See Also

`fmdemod` | `ammod` | `pmmod`

How To

- “Digital Modulation”

Purpose

Frequency shift keying demodulation

Syntax

```
z = fskdemod(y,M,freq_sep,nsamp)
z = fskdemod(y,M,freq_sep,nsamp,Fs)
z = fskdemod(y,M,freq_sep,nsamp,Fs,symbol_order)
```

Description

`z = fskdemod(y,M,freq_sep,nsamp)` noncoherently demodulates the complex envelope `y` of a signal using the frequency shift key method. `M` is the alphabet size and must be an integer power of 2. `freq_sep` is the frequency separation between successive frequencies in Hz. `nsamp` is the required number of samples per symbol and must be a positive integer greater than 1. The sampling frequency is 1 Hz. If `y` is a matrix with multiple rows and columns, the function processes the columns independently.

`z = fskdemod(y,M,freq_sep,nsamp,Fs)` specifies the sampling frequency in Hz.

`z = fskdemod(y,M,freq_sep,nsamp,Fs,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray-coded ordering.

Examples

The example below illustrates FSK modulation and demodulation over an AWGN channel.

```
M = 2; k = log2(M);
EbNo = 5;
Fs = 16; nsamp = 17; freqsep = 8;
msg = randi([0 M-1],5000,1); % Random signal
txsig = fskmod(msg,M,freqsep,nsamp,Fs); % Modulate.
msg_rx  = awgn(txsig,EbNo+10*log10(k)-10*log10(nsamp),...
    'measured',[],'dB'); % AWGN channel
msg_rrx = fskdemod(msg_rx,M,freqsep,nsamp,Fs);  % Demodulate
[num,BER] = biterr(msg,msg_rrx) % Bit error rate
BER_theory = berawgn(EbNo,'fsk',M,'noncoherent') % Theoretical BER
```

The output is shown below. Your BER value might vary because the example uses random numbers.

```
BER =

    0.1086

BER_theory =

    0.1029
```

See Also

`fskmod` | `pskmod` | `pskdemod`

How To

- "Digital Modulation"

Purpose

Frequency shift keying modulation

Syntax

```
y = fskmod(x,M,freq_sep,nsamp)
y = fskmod(x,M,freq_sep,nsamp,Fs)
y = fskmod(x,M,freq_sep,nsamp,Fs,phase_cont)
y = FSKMOD(x,M,freq_sep,nsamp,Fs,phase_cont,symbol_order)
```

Description

`y = fskmod(x,M,freq_sep,nsamp)` outputs the complex envelope `y` of the modulation of the message signal `x` using frequency shift keying modulation. `M` is the alphabet size and must be an integer power of 2. The message signal must consist of integers between 0 and `M-1`. `freq_sep` is the desired separation between successive frequencies in Hz. `nsamp` denotes the number of samples per symbol in `y` and must be a positive integer greater than 1. The sampling rate of `y` is 1 Hz. By the Nyquist sampling theorem, `freq_sep` and `M` must satisfy `(M-1)*freq_sep <= 1`. If `x` is a matrix with multiple rows and columns, the function processes the columns independently.

`y = fskmod(x,M,freq_sep,nsamp,Fs)` specifies the sampling rate of `y` in Hz. Because the Nyquist sampling theorem implies that the maximum frequency must be no larger than `Fs/2`, the inputs must satisfy `(M-1)*freq_sep <= Fs`.

`y = fskmod(x,M,freq_sep,nsamp,Fs,phase_cont)` specifies the phase continuity. Set `phase_cont` to `'cont'` to force phase continuity across symbol boundaries in `y`, or `'discont'` to avoid forcing phase continuity. The default is `'cont'`.

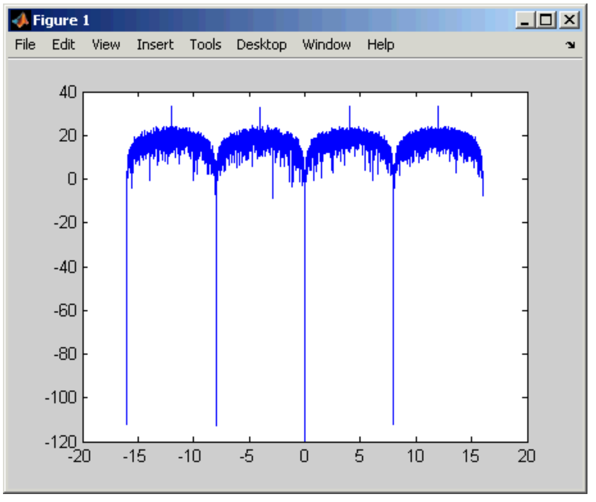
`y = FSKMOD(x,M,freq_sep,nsamp,Fs,phase_cont,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray-coded ordering.

Examples

The example below illustrates the syntax of `fskmod` using a random signal.

```
M = 4; freqsep = 8; nsamp = 8; Fs = 32;
```

```
x = randi([0 M-1],1000,1); % Random signal
y = fskmod(x,M,freqsep,nsamp,Fs); % Modulate.
ly = length(y);
% Create an FFT plot.
freq = [-Fs/2 : Fs/ly : Fs/2 - Fs/ly];
Syy = 10*log10(fftshift(abs(fft(y))));
plot(freq,Syy)
```



See Also

fskdemod | pskmod | pskdemod

How To

- “Digital Modulation”

Purpose

Convert between parity-check and generator matrices

Syntax

```
parmat = gen2par(genmat)
genmat = gen2par(parmat)
```

Description

`parmat = gen2par(genmat)` converts the standard-form binary generator matrix `genmat` into the corresponding parity-check matrix `parmat`.

`genmat = gen2par(parmat)` converts the standard-form binary parity-check matrix `parmat` into the corresponding generator matrix `genmat`.

The standard forms of the generator and parity-check matrices for an [n,k] binary linear block code are shown in the table below

Type of Matrix	Standard Form	Dimensions
Generator	$[I_k\ P]$ or $[P\ I_k]$	k-by-n
Parity-check	$[-P'\ I_{n-k}]$ or $[I_{n-k}\ -P']$	(n-k)-by-n

.

where I_k is the identity matrix of size k and the `'` symbol indicates matrix transpose. Two standard forms are listed for each type, because different authors use different conventions. For *binary* codes, the minus signs in the parity-check form listed above are irrelevant; that is, -1 = 1 in the binary field.

Examples

The commands below convert the parity-check matrix for a Hamming code into the corresponding generator matrix and back again.

```
parmat = hammgen(3)
genmat1 = gen2par(parmat)
parmat2 = gen2par(genmat1) % Ans should be the same as parmat above
```

The output is

```
parmat =
```

```
     1     0     0     1     0     1     1
     0     1     0     1     1     1     0
     0     0     1     0     1     1     1

genmat =

     1     1     0     1     0     0     0
     0     1     1     0     1     0     0
     1     1     1     0     0     1     0
     1     0     1     0     0     0     1

parmat2 =

     1     0     0     1     0     1     1
     0     1     0     1     1     1     0
     0     0     1     0     1     1     1
```

See Also

`cyclgen` | `hammgen`

How To

- "Block Codes"

Purpose

General quadrature amplitude demodulation

Syntax

`z = genqamdemod(y,const)`

Description

Warning

This function is obsolete and may be removed in the future. We strongly recommend that you use the `comm.GeneralQAMDemodulator` System object instead.

`z = genqamdemod(y,const)` demodulates the complex envelope `y` of a quadrature amplitude modulated signal. The complex vector `const` specifies the signal mapping. If `y` is a matrix with multiple rows, the function processes the columns independently.

Examples

The reference page for `genqammod` has an example that uses `genqamdemod`.

See Also

`genqammod` | `qammod` | `qamdemod` | `pammod` | `pamdemod`

How To

- "Digital Modulation"

genqammod

Purpose

General quadrature amplitude modulation

Syntax

`y = genqammod(x,const)`

Description

Warning

This function is obsolete and may be removed in the future. We strongly recommend that you use the `comm.GeneralQAMModulator` System object instead.

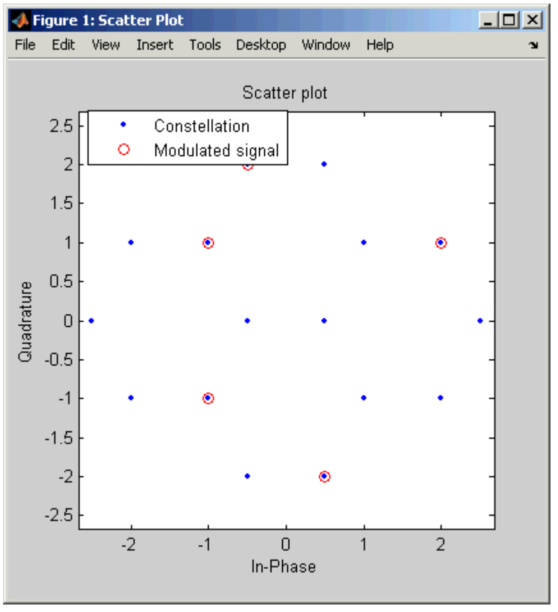
`y = genqammod(x,const)` outputs the complex envelope `y` of the modulation of the message signal `x` using quadrature amplitude modulation. The message signal must consist of integers between 0 and `length(const)-1`. The complex vector `const` specifies the signal mapping. If `x` is a matrix with multiple rows, the function processes the columns independently.

Examples

The code below plots a signal constellation that has a hexagonal structure. It also uses `genqammod` and `genqamdemod` to modulate and demodulate a message `[3 8 5 10 7]` using this constellation.

```
% Describe hexagonal constellation.
inphase = [1/2 1 1 1/2 1/2 2 2 5/2];
quadr = [0 1 -1 2 -2 1 -1 0];
inphase = [inphase;-inphase]; inphase = inphase(:);
quadr = [quadr;quadr]; quadr = quadr(:);
const = inphase + j*quadr;

% Plot constellation.
h = scatterplot(const);

% Modulate message using this constellation.
x = [3 8 5 10 7]; % Message signal
y = genqammod(x,const);
z = genqamdemod(y,const); % Demodulate.

% Plot modulated signal in same figure.
```

```
hold on; scatterplot(y,1,0,'ro',h);
legend('Constellation','Modulated signal','Location','NorthWest'); % Include legend.
hold off;
```



Another example using this function is the Gray-coded constellation example in "Examples of Signal Constellation Plots".

See Also

genqamdemod | qammod | qamdemod | pammod | pamdemod

How To

- "Digital Modulation"

Purpose Create Galois field array

Syntax

```
x_gf = gf(x,m)
x_gf = gf(x,m,prim_poly)
x_gf = gf(x)
```

Description

`x_gf = gf(x,m)` creates a Galois field array from the matrix `x`. The Galois field has `2^m` elements, where `m` is an integer between 1 and 16. The elements of `x` must be integers between 0 and `2^m-1`. The output `x_gf` is a variable that MATLAB recognizes as a Galois field array, rather than an array of integers. As a result, when you manipulate `x_gf` using operators or functions such as `+` or `det`, MATLAB works within the Galois field you have specified.

Note To learn how to manipulate `x_gf` using familiar MATLAB operators and functions, see "Galois Field Computations". To learn how the integers in `x` represent elements of GF(`2^m`), see "How Integers Correspond to Galois Field Elements".

`x_gf = gf(x,m,prim_poly)` is the same as the previous syntax, except it uses the primitive polynomial `prim_poly` to define the field. `prim_poly` is the integer representation of a primitive polynomial. For example, the number 37 represents the polynomial D^5+D^2+1 because the binary form of 37 is 1 0 0 1 0 1. For more information about the primitive polynomial, see "Specifying the Primitive Polynomial".

`x_gf = gf(x)` creates a GF(2) array from the matrix `x`. Each element of `x` must be 0 or `1`.

Default Primitive Polynomials

The table below lists the primitive polynomial that `gf` uses by default for each Galois field GF(`2^m`). To use a different primitive polynomial, specify `prim_poly` as an input argument when you invoke `gf`.

m	Default Primitive Polynomial	Integer Representation
1	D + 1	3
2	D^2 + D + 1	7
3	D^3 + D + 1	11
4	D^4 + D + 1	19
5	D^5 + D^2 + 1	37
6	D^6 + D + 1	67
7	D^7 + D^3 + 1	137
8	D^8 + D^4 + D^3 + D^2 + 1	285
9	D^9 + D^4 + 1	529
10	D^10 + D^3 + 1	1033
11	D^11 + D^2 + 1	2053
12	D^12 + D^6 + D^4 + D + 1	4179
13	D^13 + D^4 + D^3 + D + 1	8219
14	D^14 + D^10 + D^6 + D + 1	17475
15	D^15 + D + 1	32771
16	D^16 + D^12 + D^3 + D + 1	69643

Examples

For examples that use `gf`, see

- "Example: Creating Galois Field Variables"
- "Example: Representing a Primitive Element"

- Other sample code within "Galois Field Computations"
- The Galois field demonstration: type `showdemo gfdemo`.

See Also

`gftable`

How To

- Galois field computations
- "Galois Field Computations"

Purpose

Add polynomials over Galois field

Syntax

```
c = gfadd(a,b)
c = gfadd(a,b,p)
c = gfadd(a,b,p,len)
c = gfadd(a,b,field)
```

Description

Note This function performs computations in GF(p^m) where p is prime. To work in GF(2^m), apply the + operator to Galois arrays of equal size. For details, see “Example: Addition and Subtraction”.

`c = gfadd(a,b)` adds two GF(2) polynomials, `a` and `b`. If `a` and `b` are vectors of the same orientation but different lengths, then the shorter vector is zero-padded. If `a` and `b` are matrices they must be of the same size.

`c = gfadd(a,b,p)` adds two GF(`p`) polynomials, where `p` is a prime number. `a`, `b`, and `c` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p`-1. If `a` and `b` are matrices of the same size, the function treats each row independently.

`c = gfadd(a,b,p,len)` adds row vectors `a` and `b` as in the previous syntax, except that it returns a row vector of length `len`. The output `c` is a truncated or extended representation of the sum. If the row vector corresponding to the sum has fewer than `len` entries (including zeros), extra zeros are added at the end; if it has more than `len` entries, entries from the end are removed.

`c = gfadd(a,b,field)` adds two GF(p^m) elements, where `m` is a positive integer. `a` and `b` are the exponential format of the two elements, relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. `c` is the exponential format of the sum, relative to the same primitive element. See “Representing Elements of Galois Fields” for an explanation of these formats. If a and b are matrices of the same size, the function treats each element independently.

Examples

In the code below, sum5 is the sum of $2 + 3x + x^2$ and $4 + 2x + 3x^2$ over GF(5), and linpart is the degree-one part of sum5.

```
sum5 = gfadd([2 3 1],[4 2 3],5)
linpart = gfadd([2 3 1],[4 2 3],5,2)
```

The output is

```
sum5 =

     1     0     4


linpart =

     1     0
```

The code below shows that $A^2 + A^4 = A^1$, where A is a root of the primitive polynomial $2 + 2x + x^2$ for GF(9).

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
g = gfadd(2,4,field)
```

The output is

```
g =

     1
```

Other examples are in “Arithmetic in Galois Fields”.

See Also

gfsub | gfconv | gfmul | gfdeconv | gfdiv | gftuple

Purpose

Multiply polynomials over Galois field

Syntax

```
c = gfconv(a,b)
c = gfconv(a,b,p)
c = gfconv(a,b,field)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `conv` function with Galois arrays. For details, see "Multiplication and Division of Polynomials".

The `gfconv` function multiplies polynomials over a Galois field. (To multiply elements of a Galois field, use `gfmul` instead.) Algebraically, multiplying polynomials over a Galois field is equivalent to convolving vectors containing the polynomials' coefficients, where the convolution operation uses arithmetic over the same Galois field.

`c = gfconv(a,b)` multiplies two GF(2) polynomials, `a` and `b`. The polynomial degree of the resulting GF(2) polynomial `c` equals the degree of `a` plus the degree of `b`.

`c = gfconv(a,b,p)` multiplies two GF(`p`) polynomials, where `p` is a prime number. `a`, `b`, and `c` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p`-1.

`c = gfconv(a,b,field)` multiplies two GF(p^m) polynomials, where p is a prime number and m is a positive integer. `a`, `b`, and `c` are row vectors that list the exponential formats of the coefficients of the corresponding polynomials, in order of ascending powers. The exponential format is relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. See "Representing Elements of Galois Fields" for an explanation of these formats.

Examples

The command below shows that

$$(1+x+x^4)(x+x^2) = x+2x^2+x^3+x^5+x^6$$

over GF(3).

```
gfc = gfconv([1 1 0 0 1],[0 1 1],3)
```

The output is

```
gfc =

     0     1     2     1     0     1     1
```

The code below illustrates the identity

$$(x^r + x^s)^p = x^{rp} + x^{sp}$$

for the case in which $p = 7$, $r = 5$, and $s = 3$. (The identity holds when p is any prime number, and r and s are positive integers.)

```
p = 7; r = 5; s = 3;
a = gfrepcov([r s]); % x^r + x^s

% Compute a^p over GF(p).
c = 1;
for ii = 1:p
   c = gfconv(c,a,p);
end;

% Check whether c = x^(rp) + x^(sp).
powers = [];
for ii = 1:length(c)
   if c(ii)~=0
      powers = [powers, ii];
   end;
end;
if (powers==[r*p+1 s*p+1] | powers==[s*p+1 r*p+1])
```

```
   disp('The identity is proved for this case of r, s, and p.')
end
```

See Also

gfdeconv | gfadd | gfsub | gfmul | gftuple

gfcosets

Purpose

Produce cyclotomic cosets for Galois field

Syntax

```
c = gfcosets(m)
c = gfcosets(m,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `cosets` function.

`c = gfcosets(m)` produces cyclotomic cosets mod(2^m - 1). Each row of the output GFCS contains one cyclotomic coset.

`c = gfcosets(m,p)` produces the cyclotomic cosets for GF(`p^m`), where `m` is a positive integer and `p` is a prime number.

The output matrix `c` is structured so that each row represents one coset. The row represents the coset by giving the exponential format of the elements of the coset, relative to the default primitive polynomial for the field. For a description of exponential formats, see "Representing Elements of Galois Fields".

The first column contains the coset leaders. Because the lengths of cosets might vary, entries of `NaN` are used to fill the extra spaces when necessary to make `c` rectangular.

A cyclotomic coset is a set of elements that all satisfy the same minimal polynomial. For more details on cyclotomic cosets, see the works listed in "References" on page 1-287.

Examples

The command below finds the cyclotomic cosets for GF(9).

```
c = gfcosets(2,3)
```

The output is

```
c =

     0   NaN
```

```
     1     3
     2     6
     4   NaN
     5     7
```

The `gfminpol` function can check that the elements of, for example, the third row of `c` indeed belong in the same coset.

```
m = [gfminpol(2,2,3); gfminpol(6,2,3)] % Rows are identical.
```

The output is

```
m =

     1     0     1
     1     0     1
```

References

[1] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, MA, Addison-Wesley, 1983, p. 105.

[2] Lin, Shu, and Daniel J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, Englewood Cliffs, NJ, Prentice-Hall, 1983.

See Also

`gfminpol` | `gfprimdf` | `gfroots`

gfdeconv

Purpose

Divide polynomials over Galois field

Syntax

```
[quot,remd] = gfdeconv(b,a)
[quot,remd] = gfdeconv(b,a,p)
[quot,remd] = gfdeconv(b,a,field)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `deconv` function with Galois arrays. For details, see "Multiplication and Division of Polynomials".

The `gfdeconv` function divides polynomials over a Galois field. (To divide elements of a Galois field, use `gfdiv` instead.) Algebraically, dividing polynomials over a Galois field is equivalent to deconvolving vectors containing the polynomials' coefficients, where the deconvolution operation uses arithmetic over the same Galois field.

`[quot,remd] = gfdeconv(b,a)` computes the quotient `quot` and remainder `remd` of the division of `b` by `a` in GF(2).

`[quot,remd] = gfdeconv(b,a,p)` divides the polynomial `b` by the polynomial `a` over GF(`p`) and returns the quotient in `quot` and the remainder in `remd`. `p` is a prime number. `b`, `a`, `quot`, and `remd` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p`-1.

`[quot,remd] = gfdeconv(b,a,field)` divides the polynomial `b` by the polynomial `a` over GF(p^m) and returns the quotient in `quot` and the remainder in `remd`. Here p is a prime number and m is a positive integer. `b`, `a`, `quot`, and `remd` are row vectors that list the exponential formats of the coefficients of the corresponding polynomials, in order of ascending powers. The exponential format is relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. See "Representing Elements of Galois Fields" for an explanation of these formats.

Examples

The code below shows that

$$(x + x^3 + x^4) \div (1 + x) = 1 + x^3 \text{ Remainder } 2$$

in GF(3). It also checks the results of the division.

```
p = 3;
b = [0 1 0 1 1]; a = [1 1];
[quot, remd] = gfdeconv(b,a,p)
% Check the result.
bnew = gfadd(gfconv(quot,a,p),remd,p);
if isequal(bnew,b)
   disp('Correct.')
end;
```

The output is below.

```
quot =

     1     0     0     1

remd =

     2

Correct.
```

Working over GF(3), the code below outputs those polynomials of the form $x^k - 1$ (k = 2, 3, 4,..., 8) that $1 + x^2$ divides evenly.

```
p = 3; m = 2;
a = [1 0 1]; % 1+x^2
for ii = 2:p^m-1
   b = gfrepcov(ii); % x^ii
   b(1) = p-1; % -1+x^ii
   [quot, remd] = gfdeconv(b,a,p);
   % Display -1+x^ii if a divides it evenly.
   if remd==0
```

```
      multiple{ii}=b;
      gfpretty(b)
   end
end
```

The output is below.

```
         4
    2 + X

         8
    2 + X
```

In light of the discussion in “Algorithms” on page 1-303 on the `gfprimck` reference page, along with the irreducibility of $1 + x^2$ over GF(3), this output indicates that $1 + x^2$ is not primitive for GF(9).

Algorithms

The algorithm of `gfdeconv` is similar to that of the MATLAB function `deconv`.

See Also

`gfconv` | `gfadd` | `gfsub` | `gfdiv` | `gftuple`

Purpose

Divide elements of Galois field

Syntax

```
quot = gfdiv(b,a)
quot = gfdiv(b,a,p)
quot = gfdiv(b,a,field)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), apply the `./` operator to Galois arrays. For details, see "Example: Division".

The `gfdiv` function divides elements of a Galois field. (To divide polynomials over a Galois field, use `gfdeconv` instead.)

`quot = gfdiv(b,a)` divides `b` by `a` in GF(2) element-by-element. `a` and `b` are scalars, vectors or matrices of the same size. Each entry in `a` and `b` represents an element of GF(2). The entries of `a` and `b` are either 0 or 1.

`quot = gfdiv(b,a,p)` divides b by a in GF(`p`) and returns the quotient. `p` is a prime number. If `a` and `b` are matrices of the same size, the function treats each element independently. All entries of `b`, `a`, and `quot` are between 0 and `p`-1.

`quot = gfdiv(b,a,field)` divides `b` by `a` in GF(p^m) and returns the quotient. p is a prime number and m is a positive integer. If `a` and `b` are matrices of the same size, then the function treats each element independently. All entries of `b`, `a`, and `quot` are the exponential formats of elements of GF(p^m) relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. See "Representing Elements of Galois Fields" for an explanation of these formats.

In all cases, an attempt to divide by the zero element of the field results in a "quotient" of `NaN`.

Examples

The code below displays lists of multiplicative inverses in GF(5) and GF(25). It uses column vectors as inputs to `gfdiv`.

```
% Find inverses of nonzero elements of GF(5).
p = 5;
b = ones(p-1,1);
a = [1:p-1]';
quot1 = gfdiv(b,a,p);
disp('Inverses in GF(5):')
disp('element  inverse')
disp([a, quot1])

% Find inverses of nonzero elements of GF(25).
m = 2;
field = gftuple([-1:p^m-2]',m,p);
b = zeros(p^m-1,1); % Numerator is zero since 1 = alpha^0.
a = [0:p^m-2]';
quot2 = gfdiv(b,a,field);
disp('Inverses in GF(25), expressed in EXPONENTIAL FORMAT with')
disp('respect to a root of the default primitive polynomial:')
disp('element  inverse')
disp([a, quot2])
```

See Also

gfmul | gfdeconv | gfconv | gftuple

Purpose

Filter data using polynomials over prime Galois field

Syntax

```
y = gffilter(b,a,x)
y = gffilter(b,a,x,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `filter` function with Galois arrays. For details, see "Filtering".

`y = gffilter(b,a,x)` filters the data in vector `x` with the filter described by vectors `b` and `a`. The vectors `b`, `a` and `x` must be in GF(2), that is, be binary and `y` is also in GF(2).

`y = gffilter(b,a,x,p)` filters the data `x` using the filter described by vectors `a` and `b`. `y` is the filtered data in GF(`p`). `p` is a prime number, and all entries of `a` and `b` are between 0 and `p`-1.

By definition of the filter, `y` solves the difference equation

```
a(1)y(n) = b(1)x(n)+b(2)x(n-1)+b(3)x(n-2)+...+b(B+1)x(n-B)
                         -a(2)y(n-1)-a(3)y(n-2)-...-a(A+1)y(n-A)
```

where

- `A`+1 is the length of the vector `a`
- `B`+1 is the length of the vector `b`
- `n` varies between 1 and the length of the vector `x`.

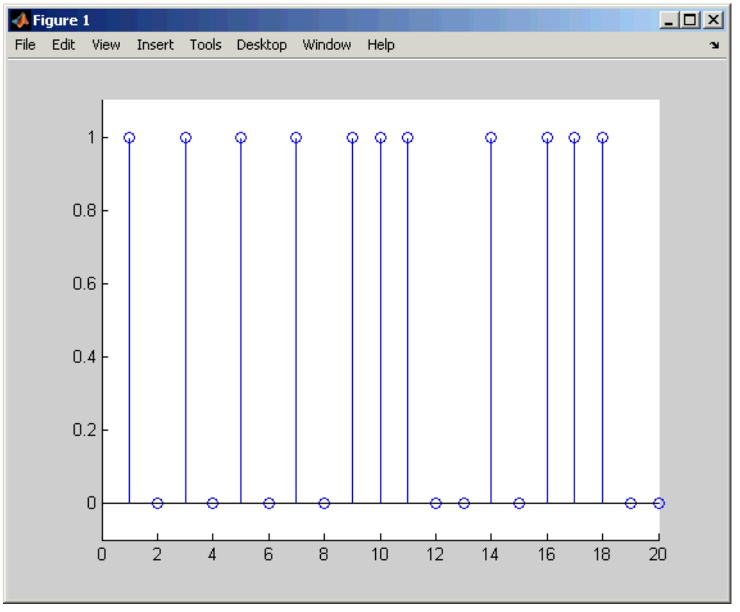
The vector `a` represents the degree-n_a polynomial

```
a(1)+a(2)x+a(3)x^2+...+a(A+1)x^A
```

Examples

The impulse response of a particular filter is given in the code and diagram below.

```
b = [1 0 0 1 0 1 0 1];
a = [1 0 1 1];
y = gffilter(b,a,[1,zeros(1,19)]);
stem(y);
axis([0 20 -.1 1.1])
```



See Also

`gfconv` | `gfadd` | `filter`

Purpose

Find particular solution of Ax = b over prime Galois field

Syntax

```
x = gflineq(A,b)
x = gflineq(A,b,p)
[x,vld] = gflineq(...)
```

Description

Note This function performs computations in GF(p), where p is prime. To work in GF(2^m), apply the \ or / operator to Galois arrays. For details, see "Solving Linear Equations".

---

x = gflineq(A,b) outputs a particular solution of the linear equation A x = b in GF(2). The elements in a, b and x are either 0 or 1. If the equation has no solution, then x is empty.

x = gflineq(A,b,p) returns a particular solution of the linear equation A x = b over GF(p), where p is a prime number. If A is a k-by-n matrix and b is a vector of length k, x is a vector of length n. Each entry of A, x, and b is an integer between 0 and p-1. If no solution exists, x is empty.

[x,vld] = gflineq(...) returns a flag vld that indicates the existence of a solution. If vld = 1, the solution x exists and is valid; if vld = 0, no solution exists.

Examples

The code below produces some valid solutions of a linear equation over GF(3).

```
A = [2 0 1;
     1 1 0;
     1 1 2];
% An example in which the solutions are valid
[x,vld] = gflineq(A,[1;0;0],3)
```

The output is below.

```
x =
```

```
     2
     1
     0


vld =

     1
```

By contrast, the command below finds that the linear equation has *no* solutions.

```
[x2,vld2] = gflineq(zeros(3,3),[2;0;0],3)
```

The output is below.

```
This linear equation has no solution.

x2 =

     []


vld2 =

     0
```

Algorithms

`gflineq` uses Gaussian elimination.

See Also

`gfadd` | `gfdiv` | `gfroots` | `gfrank` | `gfconv` | `conv`

Purpose

Find minimal polynomial of Galois field element

Syntax

```
pol = gfminpol(k,m)
pol = gfminpol(k,m,p)
pol = gfminpol(k,prim_poly,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `minpol` function with Galois arrays. For details, see "Minimal Polynomials".

`pol = gfminpol(k,m)` produces a minimal polynomial for each entry in `k`. `k` must be either a scalar or a column vector. Each entry in `k` represents an element of GF(2^m) in exponential format. That is, `k` represents alpha^`k`, where alpha is a primitive element in GF(2^m). The *i*th row of `pol` represents the minimal polynomial of `k`(*i*). The coefficients of the minimal polynomial are in the base field GF(2) and listed in order of ascending exponents.

`pol = gfminpol(k,m,p)` finds the minimal polynomial of A^k over GF(`p`), where `p` is a prime number, `m` is an integer greater than 1, and A is a root of the default primitive polynomial for GF(`p^m`). The format of the output is as follows:

- If `k` is a nonnegative integer, `pol` is a row vector that gives the coefficients of the minimal polynomial in order of ascending powers.
- If `k` is a vector of length *len* all of whose entries are nonnegative integers, `pol` is a matrix having *len* rows; the rth row of `pol` gives the coefficients of the minimal polynomial of $A^{k(r)}$ in order of ascending powers.

`pol = gfminpol(k,prim_poly,p)` is the same as the first syntax listed, except that A is a root of the primitive polynomial for GF(p^m) specified by `prim_poly`. `prim_poly` is a row vector that gives the coefficients of the degree-m primitive polynomial in order of ascending powers.

Examples

The syntax `gfminpol(k,m,p)` is used in the sample code in "Characterization of Polynomials".

See Also

`gfprimdf` | `gfcosets` | `gfroots`

Purpose

Multiply elements of Galois field

Syntax

```
c = gfmul(a,b,p)
c = gfmul(a,b,field)
```

Description

Note This function performs computations in GF(p^m) where p is prime. To work in GF(2^m), apply the `.*` operator to Galois arrays. For details, see "Example: Multiplication".

The `gfmul` function multiplies elements of a Galois field. (To multiply polynomials over a Galois field, use `gfconv` instead.)

`c = gfmul(a,b,p)` multiplies `a` and `b` in GF(p). Each entry of `a` and `b` is between 0 and `p`-1. `p` is a prime number. If `a` and `b` are matrices of the same size, the function treats each element independently.

`c = gfmul(a,b,field)` multiplies `a` and `b` in GF(p^m), where p is a prime number and m is a positive integer. `a` and `b` represent elements of GF(p^m) in exponential format relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. `c` is the exponential format of the product, relative to the same primitive element. See "Representing Elements of Galois Fields" for an explanation of these formats. If `a` and `b` are matrices of the same size, the function treats each element independently.

Examples

"Arithmetic in Galois Fields" contains examples. Also, the code below shows that

$$A^2 \cdot A^4 = A^6$$

where A is a root of the primitive polynomial $2 + 2x + x^2$ for GF(9).

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
```

```
a = gfmul(2,4,field)
```

The output is

```
a =

     6
```

See Also

gfdiv | gfdeconv | gfadd | gfsub | gftuple

Purpose

Polynomial in traditional format

Syntax

```
gfpretty(a)
gfpretty(a,st)
gfpretty(a,st,n)
```

Description

`gfpretty(a)` displays a polynomial in a traditional format, using `X` as the variable and the entries of the row vector `a` as the coefficients in order of ascending powers. The polynomial is displayed in order of ascending powers. Terms having a zero coefficient are not displayed.

`gfpretty(a,st)` is the same as the first syntax listed, except that the content of the string `st` is used as the variable instead of `X`.

`gfpretty(a,st,n)` is the same as the first syntax listed, except that the content of the string `st` is used as the variable instead of `X`, and each line of the display has width `n` instead of the default value of 79.

Note For all syntaxes: If you do not use a fixed-width font, the spacing in the display might not look correct.

Examples

Display statements about the elements of GF(81).

```
p = 3; m = 4;
ii = randi([1,p^m-2],1,1); % Random exponent for prim element
primpolys = gfprimfd(m,'all',p);
[rows, cols] = size(primpolys);
jj = randi([1,rows],1,1); % Random primitive polynomial

disp('If A is a root of the primitive polynomial')
gfpretty(primpolys(jj,:)) % Polynomial in X
disp('then the element')
gfpretty([zeros(1,ii),1],'A') % The polynomial A^ii
disp('can also be expressed as')
gfpretty(gftuple(ii,m,p),'A') % Polynomial in A
```

Below is a sample of the output.

```
If A is a root of the primitive polynomial

                                      3    4
                              2 + 2 X  + X
then the element

                                     22
                                    A
can also be expressed as

                                     2    3
                               2 + A  + A
```

See Also

gftuple | gfprimdf

Purpose

Check whether polynomial over Galois field is primitive

Syntax

```
ck = gfprimck(a)
ck = gfprimck(a,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. If you are working in GF(2^m), use the `isprimitive` function. For details, see "Finding Primitive Polynomials".

`ck = gfprimck(a)` checks whether the degree-m GF(2) polynomial `a` is a primitive polynomial for GF(2^m), where m = length(`a`) - 1. The output `ck` is as follows:

- -1 if `a` is not an irreducible polynomial
- 0 if `a` is irreducible but not a primitive polynomial for GF(p^m)
- 1 if `a` is a primitive polynomial for GF(p^m)

`ck = gfprimck(a,p)` checks whether the degree-m GF(P) polynomial `a` is a primitive polynomial for GF(p^m). p is a prime number.

This function considers the zero polynomial to be "not irreducible" and considers all polynomials of degree zero or one to be primitive.

Examples

"Characterization of Polynomials" contains examples.

Algorithms

An irreducible polynomial over GF(p) of degree at least 2 is primitive if and only if it does not divide $-1 + x^k$ for any positive integer k smaller than p^m-1.

References

[1] Clark, George C. Jr., and J. Bibb Cain, *Error-Correction Coding for Digital Communications*, New York, Plenum, 1981.

[2] Krogsgaard, K., and T., Karp, *Fast Identification of Primitive Polynomials over Galois Fields: Results from a Course Project*, ICASSP 2005, Philadelphia, PA, 2004.

See Also

gfprimfd | gfprimdf | gftuple | gfminpol | gfadd

Purpose

Provide default primitive polynomials for Galois field

Syntax

```
pol = gfprimdf(m)
pol = gfprimdf(m,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `primpoly` function. For details, see "Finding Primitive Polynomials".

`pol = gfprimdf(m)` outputs the default primitive polynomial `pol` in GF(2^m).

`pol = gfprimdf(m,p)` returns the row vector that gives the coefficients, in order of ascending powers, of the default primitive polynomial for GF(p^m). `m` is a positive integer and `p` is a prime number.

Examples

The command below shows that $2 + x + x^2$ is the default primitive polynomial for GF(5^2).

```
pol = gfprimdf(2,5)
pol =

     2     1     1
```

The code below displays the default primitive polynomial for each of the fields GF(3^m), where m ranges between 3 and 5.

```
for m = 3:5
   gfpretty(gfprimdf(m,3))
end
```

The output is below.

```
                                  3
                     1 + 2 X + X
```

```
                   4
2 + X + X

                     5
1 + 2 X + X
```

See Also

gfprimck | gfprimfd | gftuple | gfminpol

Purpose

Find primitive polynomials for Galois field

Syntax

```
pol = gfprimfd(m,opt,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `primpoly` function. For details, see "Finding Primitive Polynomials".

- If `m` = 1, `pol` = [1 1].
- A polynomial is represented as a row containing the coefficients in order of ascending powers.

`pol = gfprimfd(m,opt,p)` searches for one or more primitive polynomials for GF(`p^m`), where `p` is a prime number and `m` is a positive integer. If `m` = 1, `pol` = [1 1]. If `m` > 1, the output `pol` depends on the argument *`opt`* as shown in the table below. Each polynomial is represented in `pol` as a row containing the coefficients in order of ascending powers.

opt	Significance of pol	Format of pol
`'min'`	One primitive polynomial for GF(`p^m`) having the smallest possible number of nonzero terms	The row vector representing the polynomial
`'max'`	One primitive polynomial for GF(`p^m`) having the greatest possible number of nonzero terms	The row vector representing the polynomial

opt	Significance of pol	Format of pol
`'all'`	All primitive polynomials for GF(`p^m`)	A matrix, each row of which represents one such polynomial
A positive integer	All primitive polynomials for GF(`p^m`) that have *`opt`* nonzero terms	A matrix, each row of which represents one such polynomial

Examples

The code below seeks primitive polynomials for GF(81) having various other properties. Notice that `fourterms` is empty because no primitive polynomial for GF(81) has exactly four nonzero terms. Also notice that `fewterms` represents a *single* polynomial having three terms, while `threeterms` represents *all* of the three-term primitive polynomials for GF(81).

```
p = 3; m = 4; % Work in GF(81).
fewterms = gfprimfd(m,'min',p)
threeterms = gfprimfd(m,3,p)
fourterms = gfprimfd(m,4,p)
```

The output is below.

```
fewterms =

     2     1     0     0     1


threeterms =

     2     1     0     0     1
     2     2     0     0     1
     2     0     0     1     1
     2     0     0     2     1
```

```
No primitive polynomial satisfies the given constraints.

fourterms =

     []
```

Algorithms

`gfprimfd` tests for primitivity using `gfprimck`. If *`opt`* is `'min'`, `'max'`, or omitted, polynomials are constructed by converting decimal integers to base `p`. Based on the decimal ordering, `gfprimfd` returns the first polynomial it finds that satisfies the appropriate conditions.

See Also

`gfprimck` | `gfprimdf` | `gftuple` | `gfminpol`

gfrank

Purpose

Compute rank of matrix over Galois field

Syntax

`rk = gfrank(A,p)`

Description

Note This function performs computations in GF(p^m) where p is prime. If you are working in GF(2^m), use the `rank` function with Galois arrays. For details, see "Computing Ranks".

`rk = gfrank(A,p)` calculates the rank of the matrix `A` in GF(`p`), where `p` is a prime number.

Algorithms

`gfrank` uses an algorithm similar to Gaussian elimination.

Examples

In the code below, `gfrank` says that the matrix `A` has less than full rank. This conclusion makes sense because the determinant of `A` is zero mod `p`.

```
A = [1 0 1;
   2 1 0;
   0 1 1];
p = 3;
det_a = det(A); % Ordinary determinant of A
detmodp = rem(det(A),p); % Determinant mod p
rankp = gfrank(A,p);
disp(['Determinant = ',num2str(det_a)])
disp(['Determinant mod p is ',num2str(detmodp)])
disp(['Rank over GF(p) is ',num2str(rankp)])
```

The output is below.

```
Determinant = 3
Determinant mod p is 0
Rank over GF(p) is 2
```

Purpose

Convert one binary polynomial representation to another

Syntax

```
polystandard = gfrepcov(poly2)
```

Description

Two logical ways to represent polynomials over GF(2) are listed below.

1. `[A_0 A_1 A_2 ...  A_(m-1)]` represents the polynomial

$$\mathrm{A\_0} + \mathrm{A\_1}x + \mathrm{A\_2}x^2 + \cdots + \mathrm{A\_(m\text{-}1)}x^{m-1}$$

   Each entry `A_k` is either one or zero.

2. [A_0 A_1 A_2 ... A_(m-1)] represents the polynomial

$$x^{\mathrm{A\_0}} + x^{\mathrm{A\_1}} + x^{\mathrm{A\_2}} + \cdots + x^{\mathrm{A\_(m\text{-}1)}}$$

   Each entry `A_k` is a nonnegative integer. All entries must be distinct.

Format **1** is the standard form used by the Galois field functions in this toolbox, but there are some cases in which format **2** is more convenient.

`polystandard = gfrepcov(poly2)` converts from the second format to the first, for polynomials of degree *at least* 2. `poly2` and `polystandard` are row vectors. The entries of `poly2` are distinct integers, and at least one entry must exceed 1. Each entry of `polystandard` is either 0 or 1.

Note If `poly2` is a *binary* row vector, `gfrepcov` assumes that it is already in Format **1** above and returns it unaltered.

Examples

The command below converts the representation format of the polynomial $1 + x^2 + x^5$.

```
polystandard = gfrepcov([0 2 5])

polystandard =
```

```
     1     0     1     0     0     1
```

See Also

`gfpretty`

Purpose

Find roots of polynomial over prime Galois field

Syntax

```
rt = gfroots(f,m,p)
rt = gfroots(f,prim_poly,p)
[rt,rt_tuple] = gfroots(...)
[rt,rt_tuple,field] = gfroots(...)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `roots` function with Galois arrays. For details, see "Roots of Polynomials".

For all syntaxes, `f` is a row vector that gives the coefficients, in order of ascending powers, of a degree-d polynomial.

Note `gfroots` lists each root exactly once, ignoring multiplicities of roots.

`rt = gfroots(f,m,p)` finds roots in GF(`p^m`) of the polynomial that `f` represents. `rt` is a column vector each of whose entries is the exponential format of a root. The exponential format is relative to a root of the default primitive polynomial for GF(`p^m`).

`rt = gfroots(f,prim_poly,p)` finds roots in GF(p^m) of the polynomial that `f` represents. `rt` is a column vector each of whose entries is the exponential format of a root. The exponential format is relative to a root of the degree-m primitive polynomial for GF(p^m) that `prim_poly` represents.

`[rt,rt_tuple] = gfroots(...)` returns an additional matrix `rt_tuple`, whose kth row is the polynomial format of the root `rt(k)`. The polynomial and exponential formats are both relative to the same primitive element.

`[rt,rt_tuple,field] = gfroots(...)` returns additional matrices `rt_tuple` and `field`. `rt_tuple` is described in the preceding paragraph. `field` gives the list of elements of the extension field. The list of elements, the polynomial format, and the exponential format are all relative to the same primitive element.

Note For a description of the various formats that `gfroots` uses, see "Representing Elements of Galois Fields".

Examples

"Roots of Polynomials" contains a description and example of the use of `gfroots`.

The code below finds the polynomial format of the roots of the primitive polynomial $2 + x^3 + x^4$ for GF(81). It then displays the roots in traditional form as polynomials in `alph`. (The output is omitted here.) Because `prim_poly` is both the primitive polynomial and the polynomial whose roots are sought, `alph` itself is a root.

```
p = 3; m = 4;
prim_poly = [2 0 0 1 1]; % A primitive polynomial for GF(81)
f = prim_poly; % Find roots of the primitive polynomial.
[rt,rt_tuple] = gfroots(f,prim_poly,p);
% Display roots as polynomials in alpha.
for ii = 1:length(rt_tuple)
  gfpretty(rt_tuple(ii,:),'alpha')
end
```

See Also

`gfprimdf`

Purpose

Subtract polynomials over Galois field

Syntax

```
c = gfsub(a,b,p)
c = gfsub(a,b,p,len)
c = gfsub(a,b,field)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), apply the - operator to Galois arrays of equal size. For details, see "Example: Addition and Subtraction".

`c = gfsub(a,b,p)` calculates `a` minus `b`, where `a` and `b` represent polynomials over GF(`p`) and `p` is a prime number. `a`, `b`, and `c` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p-1`. If `a` and `b` are matrices of the same size, the function treats each row independently.

`c = gfsub(a,b,p,len)` subtracts row vectors as in the syntax above, except that it returns a row vector of length `len`. The output `c` is a truncated or extended representation of the answer. If the row vector corresponding to the answer has fewer than `len` entries (including zeros), extra zeros are added at the end; if it has more than `len` entries, entries from the end are removed.

`c = gfsub(a,b,field)` calculates `a` minus `b`, where `a` and `b` are the exponential format of two elements of GF(p^m), relative to some primitive element of GF(p^m). p is a prime number and m is a positive integer. `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. `c` is the exponential format of the answer, relative to the same primitive element. See "Representing Elements of Galois Fields" for an explanation of these formats. If `a` and `b` are matrices of the same size, the function treats each element independently.

Examples

In the code below, `differ` is the difference of $2 + 3x + x^2$ and $4 + 2x + 3x^2$ over GF(5), and `linpart` is the degree-one part of `differ`.

```
differ = gfsub([2 3 1],[4 2 3],5)
linpart = gfsub([2 3 1],[4 2 3],5,2)
```

The output is

```
differ =

     3     1     3


linpart =

     3     1
```

The code below shows that $A^2 - A^4 = A^7$, where A is a root of the primitive polynomial $2 + 2x + x^2$ for GF(9).

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
d = gfsub(2,4,field)
```

The output is

```
d =

     7
```

See Also

`gfadd` | `gfconv` | `gfmul` | `gfdeconv` | `gfdiv` | `gftuple`

Purpose

Generate file to accelerate Galois field computations

Syntax

`gftable(m,prim_poly);`

Description

`gftable(m,prim_poly)` generates a file that can help accelerate computations in the field GF(`2^m`) as described by the *nondefault* primitive polynomial `prim_poly`. The integer `m` is between 1 and 16. The integer `prim_poly` represents a primitive polynomial for GF(`2^m`) using the format described in "Specifying the Primitive Polynomial". The function places the file, called `userGftable.mat`, in your current working folder. If necessary, the function overwrites any writable existing version of the file.

Note If `prim_poly` is the default primitive polynomial for GF(`2^m`) listed in the table on the `gf` reference page, this function has no effect. A MAT-file in your MATLAB installation already includes information that facilitates computations with respect to the default primitive polynomial.

Examples

In the example below, you expect `t3` to be similar to `t1` and to be significantly smaller than `t2`, assuming that you do not already have a `userGftable.mat` file that includes the (`m`, `prim_poly`) pair (`8`, `501`). Notice that before executing the `gftable` command, MATLAB displays a warning and that after executing `gftable`, there is no warning. By executing the `gftable` command you save the GF table for faster calculations.

```
% Sample code to check how much gftable improves speed.
tic; a = gf(repmat([0:2^8-1],1000,1),8); b = a.^100; t1 = toc;
tic; a = gf(repmat([0:2^8-1],1000,1),8,501); b = a.^100; t2 = toc;
gftable(8,501); % Include this primitive polynomial in the file.
tic; a = gf(repmat([0:2^8-1],1000,1),8,501); b = a.^100; t3 = toc;
```

See Also

`gf`

How To

- “Speed and Nondefault Primitive Polynomials”

Purpose

Minimize length of polynomial representation

Syntax

```
c = gftrunc(a)
```

Description

`c = gftrunc(a)` truncates a row vector, `a`, that gives the coefficients of a GF(p) polynomial in order of ascending powers. If `a`(k) = 0 whenever k > d + 1, the polynomial has degree d. The row vector `c` omits these high-order zeros and thus has length d + 1.

Examples

In the code below, zeros are removed from the end, but *not* from the beginning or middle, of the row-vector representation of $x^2 + 2x^3 + 3x^4 + 4x^7 + 5x^8$.

```
c = gftrunc([0 0 1 2 3 0 0 4 5 0 0])
c =

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

See Also

`gfadd` | `gfsub` | `gfconv` | `gfdeconv` | `gftuple`

Purpose

Simplify or convert Galois field element formatting

Syntax

```
tp = gftuple(a,m)
tp = gftuple(a,prim_poly)
tp = gftuple(a,m,p)
tp = gftuple(a,prim_poly,p)
tp = gftuple(a,prim_poly,p,prim_ck)
[tp,expform] = gftuple(...)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To perform equivalent computations in GF(2^m), apply the `.^` operator and the `log` function to Galois arrays. For more information, see "Example: Exponentiation" and "Example: Elementwise Logarithm".

For All Syntaxes

`gftuple` serves to simplify the polynomial or exponential format of Galois field elements, or to convert from one format to another. For an explanation of the formats that `gftuple` uses, see "Representing Elements of Galois Fields".

In this discussion, the format of an element of GF(p^m) is called "simplest" if all exponents of the primitive element are

- Between 0 and m-1 for the polynomial format
- Either `-Inf`, or between 0 and p^{m-2}, for the exponential format

For all syntaxes, `a` is a matrix, each row of which represents an element of a Galois field. The format of `a` determines how MATLAB interprets it:

- If `a` is a column of integers, MATLAB interprets each row as an *exponential* format of an element. Negative integers are equivalent to `-Inf` in that they all represent the zero element of the field.
- If `a` has more than one column, MATLAB interprets each row as a *polynomial* format of an element. (Each entry of `a` must be an integer between 0 and `p`-1.)

The exponential or polynomial formats mentioned above are all relative to a primitive element specified by the *second* input argument. The second argument is described below.

For Specific Syntaxes

`tp = gftuple(a,m)` returns the simplest polynomial format of the elements that `a` represents, where the kth row of `tp` corresponds to the kth row of `a`. The formats are relative to a root of the default primitive polynomial for GF(`2^m`), where `m` is a positive integer.

`tp = gftuple(a,prim_poly)` is the same as the syntax above, except that `prim_poly` is a row vector that lists the coefficients of a degree `m` primitive polynomial for GF(`2^m`) in order of ascending exponents.

`tp = gftuple(a,m,p)` is the same as `tp = gftuple(a,m)` except that 2 is replaced by a prime number `p`.

`tp = gftuple(a,prim_poly,p)` is the same as `tp = gftuple(a,prim_poly)` except that 2 is replaced by a prime number `p`.

`tp = gftuple(a,prim_poly,p,prim_ck)` is the same as `tp = gftuple(a,prim_poly,p)` except that `gftuple` checks whether `prim_poly` represents a polynomial that is indeed primitive. If not, then `gftuple` generates an error and `tp` is not returned. The input argument `prim_ck` can be any number or string; only its existence matters.

`[tp,expform] = gftuple(...)` returns the additional matrix `expform`. The kth row of `expform` is the simplest exponential format of the element that the kth row of `a` represents. All other features are as described in earlier parts of this "Description" section, depending on the input arguments.

Examples

- "List of All Elements of a Galois Field" (end of section)
- "Converting to Simplest Polynomial Format"

As another example, the `gftuple` command below generates a list of elements of GF(`p^m`), arranged relative to a root of the default primitive polynomial. Some functions in this toolbox use such a list as an input argument.

```
p = 5; % Or any prime number
m = 4; % Or any positive integer
field = gftuple([-1:p^m-2]',m,p);
```

Finally, the two commands below illustrate the influence of the *shape* of the input matrix. In the first command, a column vector is treated as a sequence of elements expressed in exponential format. In the second command, a row vector is treated as a single element expressed in polynomial format.

```
tp1 = gftuple([0; 1],3,3)
tp2 = gftuple([0, 0, 0, 1],3,3)
```

The output is below.

```
tp1 =

     1     0     0
     0     1     0


tp2 =


     2     1     0
```

The outputs reflect that, according to the default primitive polynomial for GF(3^3), the relations below are true.

$$\alpha^0 = 1 + 0\alpha + 0\alpha^2$$

$$\alpha^1 = 0 + 1\alpha + 0\alpha^2$$

$$0 + 0\alpha + 0\alpha^2 + \alpha^3 = 2 + \alpha + 0\alpha^2$$

Algorithms

`gftuple` uses recursive callbacks to determine the exponential format.

See Also

`gfadd` | `gfmul` | `gfconv` | `gfdiv` | `gfdeconv` | `gfprimdf`

Purpose

Calculate minimum distance of linear block code

Syntax

```
wt = gfweight(genmat)
wt = gfweight(genmat,'gen')
wt = gfweight(parmat,'par')
wt = gfweight(genpoly,n)
```

Description

The minimum distance, or minimum weight, of a linear block code is defined as the smallest positive number of nonzero entries in any n-tuple that is a codeword.

`wt = gfweight(genmat)` returns the minimum distance of the linear block code whose generator matrix is `genmat`.

`wt = gfweight(genmat,'gen')` returns the minimum distance of the linear block code whose generator matrix is `genmat`.

`wt = gfweight(parmat,'par')` returns the minimum distance of the linear block code whose parity-check matrix is `parmat`.

`wt = gfweight(genpoly,n)` returns the minimum distance of the *cyclic* code whose codeword length is `n` and whose generator polynomial is represented by `genpoly`. `genpoly` is a row vector that gives the coefficients of the generator polynomial in order of ascending powers.

Examples

The commands below illustrate three different ways to compute the minimum distance of a (7,4) cyclic code.

```
n = 7;
% Generator polynomial of (7,4) cyclic code
genpoly = cyclpoly(n,4);
[parmat, genmat] = cyclgen(n,genpoly);
wts = [gfweight(genmat,'gen'),gfweight(parmat,'par'),...
    gfweight(genpoly,n)]
```

The output is

```
wts =
```

```
3     3     3
```

See Also

hammgen | cyclpoly | bchgenpoly

How To

- "Block Codes"

Purpose

Convert Gray-encoded positive integers to corresponding Gray-decoded integers

Syntax

```
y = gray2bin(x,modulation,M)
[y,map] = gray2bin(x,modulation,M)
```

Description

`y = gray2bin(x,modulation,M)` generates a Gray-decoded output vector or matrix `y` with the same dimensions as its input parameter `x`. `x` can be a scalar, vector, or matrix. `modulation` is the modulation type and must be a string equal to `'qam'`, `'pam'`, `'fsk'`, `'dpsk'`, or `'psk'`. `M` is the modulation order that can be an integer power of 2.

`[y,map] = gray2bin(x,modulation,M)` generates a Gray-decoded output `y` with its respective Gray-encoded constellation map, `map`.

You can use map output to label a Gray-encoded constellation. The map output gives the Gray encoded labels for the corresponding modulation. See the example below.

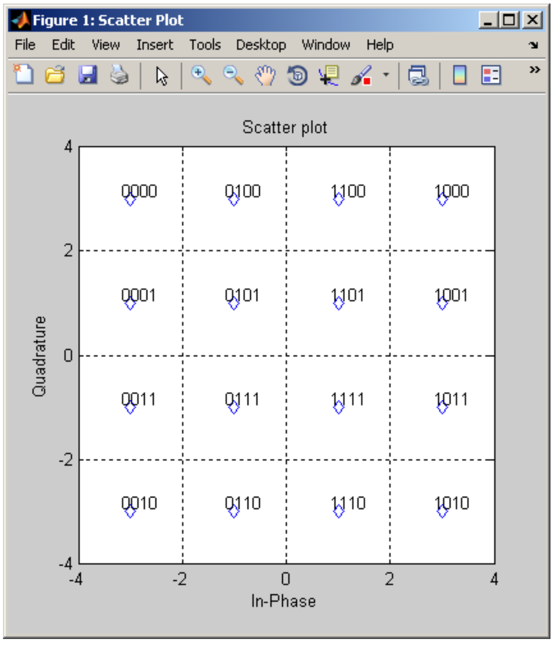
Note If you are converting binary coded data to Gray-coded data and modulating the result immediately afterwards, you should use the appropriate modulation object or function with the`'Gray'` option, instead of BIN2GRAY.

Examples

```
% To Gray decode a vector x with a 16-QAM Gray encoded
% constellation and return its map, use:
x=randi([0 15],1,100);
[y,map] = gray2bin(x,'qam',16);
% Obtain the symbols for 16-QAM
hMod = modem.qammod('M', 16);
symbols = hMod.Constellation;
% Plot the constellation
scatterplot(symbols);
set(get(gca,'Children'),'Marker','d','MarkerFaceColor','auto');
hold on;
```

```
% Label the constellation points according
% to the Gray mapping
for jj=1:16
 text(real(symbols(jj))-0.15,imag(symbols(jj))+0.15,...
 dec2base(map(jj),2,4));
end
set(gca,'yTick',(-4:2:4),'xTick',(-4:2:4),...
 'XLim',[-4 4],'YLim',...
  [-4 4],'Box','on','YGrid','on', 'XGrid','on');
```

The example code generates the following plot, which shows the 16 QAM constellation with Gray-encoded labeling.



See Also

`bin2gray`

Purpose

Produce parity-check and generator matrices for Hamming code

Syntax

```
h = hammgen(m)
h = hammgen(m,pol)
[h,g] = hammgen(...)
[h,g,n,k] = hammgen(...)
```

Description

For all syntaxes, the codeword length is `n`. `n` has the form 2^m-1 for some positive integer m greater than or equal to 3. The message length, `k`, has the form `n`-m.

`h = hammgen(m)` produces an `m`-by-n parity-check matrix for a Hamming code having codeword length `n = 2^m-1`. The input `m` is a positive integer greater than or equal to 3. The message length of the code is n-`m`. The binary primitive polynomial used to produce the Hamming code is the default primitive polynomial for GF(`2^m`), represented by `gfprimdf(m)`.

`h = hammgen(m,pol)` produces an `m`-by-n parity-check matrix for a Hamming code having codeword length `n = 2^m-1`. The input `m` is a positive integer greater than or equal to 3. The message length of the code is n-`m`. `pol` is a row vector that gives the coefficients, in order of ascending powers, of the binary primitive polynomial for GF(`2^m`) that is used to produce the Hamming code. `hammgen` produces an error if `pol` represents a polynomial that is not, in fact, primitive.

`[h,g] = hammgen(...)` is the same as `h = hammgen(...)` except that it also produces the `k`-by-n generator matrix `g` that corresponds to the parity-check matrix `h`. `k`, the message length, equals n`-m`, or `2^m-1-m`.

`[h,g,n,k] = hammgen(...)` is the same as `[h,g] = hammgen(...)` except that it also returns the codeword length `n` and the message length `k`.

Note If your value of `m` is less than 25 and if your primitive polynomial is the default primitive polynomial for GF(`2^m`), the syntax `hammgen(m)` is likely to be faster than the syntax `hammgen(m,pol)`.

Examples

The command below exhibits the parity-check and generator matrices for a Hamming code with codeword length $7 = 2^3-1$ and message length $4 = 7-3$.

```
[h,g,n,k] = hammgen(3)

h =

     1     0     0     1     0     1     1
     0     1     0     1     1     1     0
     0     0     1     0     1     1     1


g =

     1     1     0     1     0     0     0
     0     1     1     0     1     0     0
     1     1     1     0     0     1     0
     1     0     1     0     0     0     1


n =

     7


k =

     4
```

The command below, which uses $1 + x^2 + x^3$ as the primitive polynomial for $GF(2^3)$, shows that the parity-check matrix depends on the choice of primitive polynomial. Notice that `h1` below is different from `h` in the example above.

```
h1 = hammgen(3,[1 0 1 1])
```

```
h1 =

     1     0     0     1     1     1     0
     0     1     0     0     1     1     1
     0     0     1     1     1     0     1
```

Algorithms

Unlike `gftuple`, which processes one `m`-tuple at a time, `hammgen` generates the entire sequence from 0 to `2^m-1`. The computation algorithm uses all previously computed values to produce the computation result.

See Also

`encode` | `decode` | `gen2par`

How To

- "Block Codes"

hank2sys

Purpose

Convert Hankel matrix to linear system model

Syntax

```
[num,den] = hank2sys(h,ini,tol)
[num,den,sv] = hank2sys(h,ini,tol)
[a,b,c,d] = hank2sys(h,ini,tol)
[a,b,c,d,sv] = hank2sys(h,ini,tol)
```

Description

`[num,den] = hank2sys(h,ini,tol)` converts a Hankel matrix `h` to a linear system transfer function with numerator `num` and denominator `den`. The vectors `num` and `den` list the coefficients of their respective polynomials in ascending order of powers of z^{-1}. The argument `ini` is the system impulse at time zero. If `tol` > 1, `tol` is the order of the conversion. If `tol` < 1, `tol` is the tolerance in selecting the conversion order based on the singular values. If you omit `tol`, its default value is 0.01. This conversion uses the singular value decomposition method.

`[num,den,sv] = hank2sys(h,ini,tol)` returns a vector `sv` that lists the singular values of `h`.

`[a,b,c,d] = hank2sys(h,ini,tol)` converts a Hankel matrix `h` to a corresponding linear system state-space model. `a`, `b`, `c`, and `d` are matrices. The input parameters are the same as in the first syntax above.

`[a,b,c,d,sv] = hank2sys(h,ini,tol)` is the same as the syntax above, except that `sv` is a vector that lists the singular values of `h`.

Examples

```
h = hankel([1 0 1]);
[num,den,sv] = hank2sys(h,0,.01)
```

The output is

```
num =

         0    1.0000    0.0000    1.0000


den =
```

```
    1.0000    0.0000    0.0000    0.0000


sv =

    1.6180
    1.0000
    0.6180
```

See Also

rcosflt | hankel

heldeintrlv

Purpose

Restore ordering of symbols permuted using `helintrlv`

Syntax

```
[deintrlved,state] = heldeintrlv(data,col,ngrp,stp)
[deintrlved,state] = heldeintrlv(data,col,ngrp,stp,
   init_state)
deintrlved = heldeintrlv(data,col,ngrp,stp,init_state)
```

Description

`[deintrlved,state] = heldeintrlv(data,col,ngrp,stp)` restores the ordering of symbols in `data` by placing them in an array row by row and then selecting groups in a helical fashion to place in the output, `deintrlved`. `data` must have `col*ngrp` elements. If `data` is a matrix with multiple rows and columns, it must have `col*ngrp` rows, and the function processes the columns independently. `state` is a structure that holds the final state of the array. `state.value` stores input symbols that remain in the `col` columns of the array and do not appear in the output.

The function uses the array internally for its computations. The array has unlimited rows indexed by 1, 2, 3,..., and `col` columns. The function initializes the top of the array with zeros. It then places `col*ngrp` symbols from the input into the next `ngrp` rows of the array. The function places symbols from the array in the output, `intrlved`, placing `ngrp` symbols at a time; the kth group of `ngrp` symbols comes from the kth column of the array, starting from row 1+(k-1)*`stp`. Some output symbols are default values of 0 rather than input symbols; similarly, some input symbols are left in the array and do not appear in the output.

`[deintrlved,state] = heldeintrlv(data,col,ngrp,stp,init_state)` initializes the array with the symbols contained in `init_state.value` instead of zeros. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding interleaver. In this syntax, some output symbols are default values of 0, some are input symbols from `data`, and some are initialization values from `init_state.value`.

`deintrlved = heldeintrlv(data,col,ngrp,stp,init_state)` is the same as the syntax above, except that it does not record the deinterleaver's final state. This syntax is appropriate for the last in a series of calls to this function. However, if you plan to call this function again to continue the deinterleaving process, the syntax above is more appropriate.

Using an Interleaver-Deinterleaver Pair

To use this function as an inverse of the `helintrlv` function, use the same `col`, `ngrp`, and `stp` inputs in both functions. In that case, the two functions are inverses in the sense that applying `helintrlv` followed by `heldeintrlv` leaves data unchanged, after you take their combined delay of `col*ngrp*ceil(stp*(col-1)/ngrp)` into account. To learn more about delays of convolutional interleavers, see "Delays of Convolutional Interleavers".

Note Because the delay is an integer multiple of the number of symbols in `data`, you must use `heldeintrlv` at least *twice* (possibly more times, depending on the actual delay value) before the function returns results that represent more than just the delay.

Examples

Recover interleaved data, taking into account the delay of the interleaver-deinterleaver pair.

```
col = 4; ngrp = 3; stp = 2; % Helical interleaver parameters
% Compute the delay of interleaver-deinterleaver pair.
delayval = col * ngrp * ceil(stp * (col-1)/ngrp);

len = col*ngrp; % Process this many symbols at one time.
data = randi([0 9],len,1); % Random symbols
data_padded = [data; zeros(delayval,1)]; % Pad with zeros.

% Interleave zero-padded data.
[i1,istate] = helintrlv(data_padded(1:len),col,ngrp,stp);
[i2,istate] = helintrlv(data_padded(len+1:2*len),col,ngrp, ...
              stp,istate);
i3 = helintrlv(data_padded(2*len+1:end),col,ngrp,stp,istate);
```

```
% Deinterleave.
[d1,dstate] = heldeintrlv(i1,col,ngrp,stp);
[d2,dstate] = heldeintrlv(i2,col,ngrp,stp,dstate);
d3 = heldeintrlv(i3,col,ngrp,stp,dstate);

% Check the results.
d0 = [d1; d2; d3]; % All the deinterleaved data
d0_trunc = d0(delayval+1:end); % Remove the delay.
ser = symerr(data,d0_trunc)
```

The output below shows that no symbol errors occurred.

```
ser =

     0
```

See Also

`helintrlv`

How To

- "Interleaving"

Purpose

Permute symbols using helical array

Syntax

```
intrlved = helintrlv(data,col,ngrp,stp)
[intrlved,state] = helintrlv(data,col,ngrp,stp)
[intrlved,state] = helintrlv(data,col,ngrp,stp,init_state)
```

Description

`intrlved = helintrlv(data,col,ngrp,stp)` permutes the symbols in `data` by placing them in an unlimited-row array in helical fashion and then placing rows of the array in the output, `intrlved`. `data` must have `col*ngrp` elements. If `data` is a matrix with multiple rows and columns, it must have `col*ngrp` rows, and the function processes the columns independently.

The function uses the array internally for its computations. The array has unlimited rows indexed by 1, 2, 3,..., and `col` columns. The function partitions `col*ngrp` symbols from the input into consecutive groups of `ngrp` symbols. The function places the kth group in the array along column k, starting from row 1+(k-1)*`stp`. Positions in the array that do not contain input symbols have default values of 0. The function places `col*ngrp` symbols from the array in the output, `intrlved`, by reading the first `ngrp` rows sequentially. Some output symbols are default values of 0 rather than input symbols; similarly, some input symbols are left in the array and do not appear in the output.

`[intrlved,state] = helintrlv(data,col,ngrp,stp)` returns a structure that holds the final state of the array. `state.value` stores input symbols that remain in the `col` columns of the array and do not appear in the output.

`[intrlved,state] = helintrlv(data,col,ngrp,stp,init_state)` initializes the array with the symbols contained in `init_state.value`. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding deinterleaver. In this syntax, some output symbols are default values of 0, some are input symbols from `data`, and some are initialization values from `init_state.value`.

Examples

The example below rearranges the integers from 1 to 24.

```
% Interleave some symbols. Record final state of array.
[i1,state] = helintrlv([1:12]',3,4,1);
% Interleave more symbols, remembering the symbols that
% were left in the array from the earlier command.
i2 = helintrlv([13:24]',3,4,1,state);

disp('Interleaved data:')
disp([i1,i2]')
disp('Values left in array after first interleaving operation:')
state.value{:}
```

During the successive calls to `helintrlv`, it internally creates the three-column arrays

```
[1  0  0;
 2  5  0;
 3  6  9;
 4  7 10;
 0  8 11;
 0  0 12]
```

and

```
[13  8 11;
 14 17 12;
 15 18 21;
 16 19 22;
  0 20 23;
  0  0 24]
```

In the second array shown above, the 8, 11, and 12 are values left in the array from the previous call to the function. Specifying the `init_state` input in the second call to the function causes it to use those values rather than the default values of 0.

The output from this example is below. (The actual interleaved data is a tall matrix, but it has been transposed into a wide matrix for display purposes.) The interleaved data comes from the top four rows of the three-column arrays shown above. Notice that some of the symbols in the first half of the interleaved data are default values of 0, some of the symbols in the second half of the interleaved data were left in the array from the first call to `helintrlv`, and some of the input symbols (20, 23, and 24) do not appear in the interleaved data at all.

```
Interleaved data:
  Columns 1 through 10

     1     0     0     2     5     0     3     6     9     4
    13     8    11    14    17    12    15    18    21    16

  Columns 11 through 12

     7    10
    19    22

Values left in array after first interleaving operation:

ans =

     []


ans =

     8


ans =

    11    12
```

The example on the reference page for `heldeintrlv` also uses this function.

See Also

`heldeintrlv`

How To

- “Interleaving”

Purpose

Restore ordering of symbols in helical pattern

Syntax

```
deintrlvd = helscandeintrlv(data,Nrows,Ncols,hstep)
```

Description

`deintrlvd = helscandeintrlv(data,Nrows,Ncols,hstep)` rearranges the elements in `data` by filling a temporary matrix with the elements in a helical fashion and then sending the matrix contents to the output row by row. `Nrows` and `Ncols` are the dimensions of the temporary matrix. `hstep` is the slope of the diagonal, that is, the amount by which the row index increases as the column index increases by one. `hstep` must be a nonnegative integer less than `Nrows`.

Helical fashion means that the function places input elements along diagonals of the temporary matrix. The number of elements in each diagonal is exactly `Ncols`, after the function wraps past the edges of the matrix when necessary. The function traverses diagonals so that the row index and column index both increase. Each diagonal after the first one begins one row below the first element of the previous diagonal.

If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

To use this function as an inverse of the `helscanintrlv` function, use the same `Nrows`, `Ncols`, and `hstep` inputs in both functions. In that case, the two functions are inverses in the sense that applying `helscanintrlv` followed by `helscandeintrlv` leaves data unchanged.

Examples

The command below rearranges a vector using a 3-by-4 temporary matrix and diagonals of slope 1.

```
d = helscandeintrlv(1:12,3,4,1)
d =

  Columns 1 through 10

     1    10     7     4     5     2    11     8     9     6
```

```
  Columns 11 through 12

     3    12
```

Internally, the function creates the 3-by-4 temporary matrix

```
[1 10  7  4;
 5  2 11  8;
 9  6  3 12]
```

using length-four diagonals. The function then sends the elements, row by row, to the output d.

See Also

helscanintrlv

How To

- "Interleaving"

Purpose

Reorder symbols in helical pattern

Syntax

```
intrlvd = helscanintrlv(data,Nrows,Ncols,hstep)
```

Description

`intrlvd = helscanintrlv(data,Nrows,Ncols,hstep)` rearranges the elements in `data` by filling a temporary matrix with the elements row by row and then sending the matrix contents to the output in a helical fashion. `Nrows` and `Ncols` are the dimensions of the temporary matrix. `hstep` is the slope of the diagonal, that is, the amount by which the row index increases as the column index increases by one. `hstep` must be a nonnegative integer less than `Nrows`.

Helical fashion means that the function selects elements along diagonals of the temporary matrix. The number of elements in each diagonal is exactly `Ncols`, after the function wraps past the edges of the matrix when necessary. The function traverses diagonals so that the row index and column index both increase. Each diagonal after the first one begins one row below the first element of the previous diagonal.

If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

Examples

The command below rearranges a vector using diagonals of two different slopes.

```
i1 = helscanintrlv(1:12,3,4,1) % Slope of diagonal is 1.
i2 = helscanintrlv(1:12,3,4,2) % Slope of diagonal is 2.
```

The output is below.

```
i1 =

  Columns 1 through 10

     1     6    11     4     5    10     3     8     9     2

  Columns 11 through 12
```

```
     7    12


i2 =

  Columns 1 through 10

     1    10     7     4     5     2    11     8     9     6

  Columns 11 through 12

     3    12
```

In each case, the function internally creates the temporary 3-by-4 matrix

```
[1  2  3  4;
 5  6  7  8;
 9 10 11 12]
```

To form `i1`, the function forms each slope-one diagonal by moving one row down and one column to the right. The first diagonal contains `1`, `6`, `11`, and `4`, while the second diagonal starts with `5` because that is beneath `1` in the temporary matrix.

To form `i2`, the function forms each slope-two diagonal by moving two rows down and one column to the right. The first diagonal contains `1`, `10`, `7`, and `4`, while the second diagonal starts with `5` because that is beneath `1` in the temporary matrix.

See Also

`helscandeintrlv`

How To

- "Interleaving"

Purpose

Design Hilbert transform IIR filter

Syntax

```
hilbiir
hilbiir(ts)
hilbiir(ts,dly)
hilbiir(ts,dly,bandwidth)
hilbiir(ts,dly,bandwidth,tol)
[num,den] = hilbiir(...)
[num,den,sv] = hilbiir(...)
[a,b,c,d] = hilbiir(...)
[a,b,c,d,sv] = hilbiir(...)
```

Description

The function `hilbiir` designs a Hilbert transform filter. The output is either

- A plot of the filter's impulse response, or
- A quantitative characterization of the filter, using either a transfer function model or a state-space model

Background Information

An ideal Hilbert transform filter has the transfer function `H(s) = -jsgn(s)`, where `sgn(.)` is the signum function (`sign` in MATLAB). The impulse response of the Hilbert transform filter is

$$h(t) = \frac{1}{\pi t}$$

Because the Hilbert transform filter is a noncausal filter, the `hilbiir` function introduces a group delay, `dly`. A Hilbert transform filter with this delay has the impulse response

$$h(t) = \frac{1}{\pi(t - \mathrm{dly})}$$

Choosing a Group Delay Parameter

The filter design is an approximation. If you provide the filter's group delay as an input argument, these two suggestions can help improve the accuracy of the results:

- Choose the sample time `ts` and the filter's group delay `dly` so that `dly` is at least a few times larger than `ts` and `rem(dly,ts) = ts/2`. For example, you can set `ts` to 2*`dly/N`, where `N` is a positive integer.
- At the point t = `dly`, the impulse response of the Hilbert transform filter can be interpreted as `0`, `-Inf`, or `Inf`. If `hilbiir` encounters this point, it sets the impulse response there to zero. To improve accuracy, avoid the point t = `dly`.

Syntaxes for Plots

Each of these syntaxes produces a plot of the impulse response of the filter that the `hilbiir` function designs, as well as the impulse response of a corresponding ideal Hilbert transform filter.

`hilbiir` plots the impulse response of a fourth-order digital Hilbert transform filter with a one-second group delay. The sample time is 2/7 seconds. In this particular design, the tolerance index is 0.05. The plot also displays the impulse response of the ideal Hilbert transform filter with a one-second group delay.

`hilbiir(ts)` plots the impulse response of a fourth-order Hilbert transform filter with a sample time of `ts` seconds and a group delay of `ts*7/2` seconds. The tolerance index is 0.05. The plot also displays the impulse response of the ideal Hilbert transform filter having a sample time of `ts` seconds and a group delay of `ts*7/2` seconds.

`hilbiir(ts,dly)` is the same as the syntax above, except that the filter's group delay is `dly` for both the ideal filter and the filter that `hilbiir` designs. See "Choosing a Group Delay Parameter" on page 1-344 above for guidelines on choosing `dly`.

`hilbiir(ts,dly,bandwidth)` is the same as the syntax above, except that `bandwidth` specifies the assumed bandwidth of the input signal and that the filter design might use a compensator for the input signal.

If `bandwidth` = 0 or `bandwidth` > 1/(2*`ts`), `hilbiir` does not use a compensator.

`hilbiir(ts,dly,bandwidth,tol)` is the same as the syntax above, except that `tol` is the tolerance index. If `tol` < 1, the order of the filter is determined by

$$\frac{\text{truncated-singular-value}}{\text{maximum-singular-value}} < \texttt{tol}$$

If `tol` > 1, the order of the filter is `tol`.

Syntaxes for Transfer Function and State-Space Quantities

Each of these syntaxes produces quantitative information about the filter that `hilbiir` designs, but does *not* produce a plot. The input arguments for these syntaxes (if you provide any) are the same as those described in "Syntaxes for Plots" on page 1-344.

`[num,den] = hilbiir(...)` outputs the numerator and denominator of the IIR filter's transfer function.

`[num,den,sv] = hilbiir(...)` outputs the numerator and denominator of the IIR filter's transfer function, and the singular values of the Hankel matrix that `hilbiir` uses in the computation.

`[a,b,c,d] = hilbiir(...)` outputs the discrete-time state-space model of the designed Hilbert transform filter. `a`, `b`, `c`, and `d` are matrices.

`[a,b,c,d,sv] = hilbiir(...)` outputs the discrete-time state-space model of the designed Hilbert transform filter, and the singular values of the Hankel matrix that `hilbiir` uses in the computation.

Algorithms

The `hilbiir` function calculates the impulse response of the ideal Hilbert transform filter response with a group delay. It fits the response curve using a singular-value decomposition method. See the book by Kailath [1].

Examples

For an example using the function's default values, type one of the following commands at the MATLAB prompt.

```
hilbiir
[num,den] = hilbiir
```

References

[1] Kailath, Thomas, *Linear Systems*, Englewood Cliffs, NJ, Prentice-Hall, 1980.

See Also

`grpdelay` | `rcosiir`

How To

- "Filtering"

Purpose

Huffman decoder

Syntax

```
dsig = huffmandeco(comp,dict)
```

Description

`dsig = huffmandeco(comp,dict)` decodes the numeric Huffman code vector `comp` using the code dictionary `dict`. The argument `dict` is an N-by-2 cell array, where N is the number of distinct possible symbols in the original signal that was encoded as `comp`. The first column of `dict` represents the distinct symbols and the second column represents the corresponding codewords. Each codeword is represented as a numeric row vector, and no codeword in `dict` is allowed to be the prefix of any other codeword in `dict`. You can generate `dict` using the `huffmandict` function and `comp` using the `huffmanenco` function. If all signal values in `dict` are numeric, `dsig` is a vector; if any signal value in `dict` is alphabetical, `dsig` is a one-dimensional cell array.

Examples

The example below encodes and then decodes a vector of random data that has a prescribed probability distribution.

```
symbols = [1:6]; % Distinct symbols that data source can produce
p = [.5 .125 .125 .125 .0625 .0625]; % Probability distribution
[dict,avglen] = huffmandict(symbols,p); % Create dictionary.
actualsig = randsrc(1,100,[symbols; p]); % Create data using p.
comp = huffmanenco(actualsig,dict); % Encode the data.
dsig = huffmandeco(comp,dict); % Decode the Huffman code.
isequal(actualsig,dsig) % Check whether the decoding is correct.
```

The output below indicates that the decoder successfully recovered the data in `actualsig`.

```
ans =

     1
```

References

[1] Sayood, Khalid, *Introduction to Data Compression,* San Francisco, Morgan Kaufmann, 2000.

See Also

`huffmandict` | `huffmanenco`

How To

- “Huffman Coding”

Purpose

Generate Huffman code dictionary for source with known probability model

Syntax

```
[dict,avglen] = huffmandict(symbols,p)
[dict,avglen] = huffmandict(symbols,p,N)
[dict,avglen] = huffmandict(symbols,p,N,variance)
```

Description

For All Syntaxes

The `huffmandict` function generates a Huffman code dictionary corresponding to a source with a known probability model. The required inputs are

- `symbols`, which lists the distinct signal values that the source produces. It can have the form of a numeric vector, numeric cell array, or alphanumeric cell array. If it is a cell array, it must be either a row or a column.
- `p`, a probability vector whose kth element is the probability with which the source produces the kth element of `symbols`. The length of `p` must equal the length of `symbols`.

The outputs of `huffmandict` are

- `dict`, a two-column cell array in which the first column lists the distinct signal values from `symbols` and the second column lists the corresponding Huffman codewords. In the second column, each Huffman codeword is represented as a numeric row vector.
- `avglen`, the average length among all codewords in the dictionary, weighted according to the probabilities in the vector `p`.

For Specific Syntaxes

`[dict,avglen] = huffmandict(symbols,p)` generates a binary Huffman code dictionary using the maximum variance algorithm.

`[dict,avglen] = huffmandict(symbols,p,N)` generates an `N`-ary Huffman code dictionary using the maximum variance algorithm. `N` is an integer between 2 and 10 that must not exceed the number of source symbols whose probabilities appear in the vector `p`.

`[dict,avglen] = huffmandict(symbols,p,N,variance)` generates an N-ary Huffman code dictionary with the minimum variance if *`variance`* is `'min'` and the maximum variance if *`variance`* is `'max'`. `N` is an integer between 2 and 10 that must not exceed the length of the vector `p`.

Examples

```
symbols = [1:5];
p = [.3 .3 .2 .1 .1];
[dict,avglen] = huffmandict(symbols,p)
samplecode = dict{5,2} % Codeword for fifth signal value
```

The output is below, where the first column of `dict` lists the values in `symbols` and the second column lists the corresponding codewords.

```
dict =

    [1]    [1x2 double]
    [2]    [1x2 double]
    [3]    [1x2 double]
    [4]    [1x3 double]
    [5]    [1x3 double]


avglen =

    2.2000


samplecode =

     1     1     0
```

References

[1] Sayood, Khalid, *Introduction to Data Compression,* San Francisco, Morgan Kaufmann, 2000.

See Also

`huffmanenco` | `huffmandeco`

How To

- "Huffman Coding"

huffmanenco

Purpose

Huffman encoder

Syntax

```
comp = huffmanenco(sig,dict)
```

Description

`comp = huffmanenco(sig,dict)` encodes the signal `sig` using the Huffman codes described by the code dictionary `dict`. The argument `sig` can have the form of a numeric vector, numeric cell array, or alphanumeric cell array. If `sig` is a cell array, it must be either a row or a column. `dict` is an N-by-2 cell array, where N is the number of distinct possible symbols to be encoded. The first column of `dict` represents the distinct symbols and the second column represents the corresponding codewords. Each codeword is represented as a numeric row vector, and no codeword in `dict` can be the prefix of any other codeword in `dict`. You can generate `dict` using the `huffmandict` function.

Examples

The example below encodes a vector of random data that has a prescribed probability distribution.

```
symbols = [1:6]; % Distinct symbols that data source can produce
p = [.5 .125 .125 .125 .0625 .0625]; % Probability distribution
[dict,avglen] = huffmandict(symbols,p); % Create dictionary.
actualsig = randsrc(100,1,[symbols; p]); % Create data using p.
comp = huffmanenco(actualsig,dict); % Encode the data.
```

References

[1] Sayood, Khalid, *Introduction to Data Compression,* San Francisco, Morgan Kaufmann, 2000.

See Also

`huffmandict` | `huffmandeco`

How To

- “Huffman Coding”

Purpose

Inverse discrete Fourier transform

Syntax

`ifft(x)`

Description

`ifft(x)` is the inverse discrete Fourier transform (DFT) of the Galois vector `x`. If `x` is in the Galois field GF(2^m), the length of `x` must be 2^m-1.

Examples

For an example using `ifft`, see the reference page for `fft`.

Limitations

The Galois field over which this function works must have 256 or fewer elements. In other words, `x` must be in the Galois field GF(2^m), where m is an integer between 1 and 8.

Algorithms

If `x` is a column vector, `ifft` applies `dftmtx` to the multiplicative inverse of the primitive element of the Galois field and multiplies the resulting matrix by `x`.

See Also

`fft` | `dftmtx`

How To

- “Signal Processing Operations in Galois Fields”

intdump

Purpose

Integrate and dump

Syntax

```
y = intdump(x,nsamp)
```

Description

`y = intdump(x,nsamp)` integrates the signal `x` for one symbol period, then outputs the averaged one value into `Y`. `nsamp` is the number of samples per symbol. For two-dimensional signals, the function treats each column as one channel.

Examples

An example in "Combine Pulse Shaping and Filtering with Modulation" uses this function in conjunction with modulation.

Processes two independent channels, each of which contain three symbols of data made up of four samples.

```
s = rng;
rng(68521);
nsamp = 4; % Number of samples per symbol
ch1 = randi([0 1],3*nsamp,1); % Random binary channel
ch2 = rectpulse([1 2 3]',nsamp); % Rectangular pulses
x = [ch1 ch2]; % Two-channel signal
y = intdump(x,nsamp)
rng(s);
```

The output is below. Each column corresponds to one channel, and each row corresponds to one symbol.

```
y =

    0.5000    1.0000
    0.5000    2.0000
    1.0000    3.0000
```

See Also

`rectpulse`

Purpose

Reorder sequence of symbols

Syntax

```
intrlvd = intrlv(data,elements)
```

Description

`intrlvd = intrlv(data,elements)` rearranges the elements of `data` without repeating or omitting any elements. If `data` is a length-N vector or an N-row matrix, `elements` is a length-N vector that permutes the integers from 1 to N. The sequence in `elements` is the sequence in which elements from `data` or its columns appear in `intrlvd`. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

Examples

The command below rearranges the elements of a vector. Your output might differ because the permutation vector is random in this example.

```
p = randperm(10); % Permutation vector
a = intrlv(10:10:100,p)
```

The output is below.

```
a =

    10    90    60    30    50    80   100    20    70    40
```

The command below rearranges each of two columns of a matrix.

```
b = intrlv([.1 .2 .3 .4 .5; .2 .4 .6 .8 1]',[2 4 3 5 1])
b =

    0.2000    0.4000
    0.4000    0.8000
    0.3000    0.6000
    0.5000    1.0000
    0.1000    0.2000
```

See Also

`deintrlv`

How To

- “Interleaving”

Purpose

True for trellis corresponding to catastrophic convolutional code

Syntax

```
iscatastrophic(s)
```

Description

`iscatastrophic(s)` returns `true` if the trellis `s` corresponds to a convolutional code that causes catastrophic error propagation. Otherwise, it returns `false`.

See Also

`convenc` | `istrellis` | `poly2trellis` | `struct`

How To

- "Convolutional Codes"

Purpose

True for primitive polynomial for Galois field

Syntax

```
isprimitive(a)
```

Description

`isprimitive(a)` returns `1` if the polynomial that `a` represents is primitive for the Galois field GF(2^m), and `0` otherwise. The input `a` can represent the polynomial using one of these formats:

- A nonnegative integer less than 2^{17}. The binary representation of this integer indicates the coefficients of the polynomial. In this case, m is `floor(log2(a))`.
- A Galois row vector in GF(2), listing the coefficients of the polynomial in order of descending powers. In this case, m is the order of the polynomial represented by `a`.

Examples

The example below finds all primitive polynomials for GF(8) and then checks using `isprimitive` whether specific polynomials are primitive.

```
a = primpoly(3,'all','nodisplay'); % All primitive polys for GF(8)

isp1 = isprimitive(13) % 13 represents a primitive polynomial.

isp2 = isprimitive(14) % 14 represents a nonprimitive polynomial.
```

The output is below. If you examine the vector `a`, notice that `isp1` is true because `13` is an element in `a`, while `isp2` is false because `14` is not an element in `a`.

```
isp1 =

     1


isp2 =

     0
```

See Also

`primpoly`

How To

- “Galois Field Computations”

Purpose

True for valid trellis structure

Syntax

```
[isok,status] = istrellis(s)
```

Description

`[isok,status] = istrellis(s)` checks if the input `s` is a valid trellis structure. If the input is a valid trellis structure, `isok` is 1 and `status` is an empty string. Otherwise, `isok` is 0 and `status` is a string that indicates why `s` is not a valid trellis structure.

A valid trellis structure is a MATLAB structure whose fields are as in the table below.

Fields of a Valid Trellis Structure for a Rate k/n Code

Field in Trellis Structure	Dimensions	Meaning
`numInputSymbols`	Scalar	Number of input symbols to the encoder: 2^k
`numOutputSymbols`	Scalar	Number of output symbols from the encoder: 2^n
`numStates`	Scalar	Number of states in the encoder
`nextStates`	`numStates`-by-2^k matrix	Next states for all combinations of current state and current input
`outputs`	`numStates`-by-2^k matrix	Outputs (in octal) for all combinations of current state and current input

In the nextStates matrix, each entry is an integer between 0 and numStates-1. The element in the sth row and uth column denotes the next state when the starting state is s-1 and the input bits have decimal representation u-1. To convert the input bits to a decimal value, use the first input bit as the most significant bit (MSB). For example, the second column of the nextStates matrix stores the next states when the current set of input values is {0,...,0,1}.

To convert the state to a decimal value, use this rule: If k exceeds 1, the shift register that receives the first input stream in the encoder provides the least significant bits in the state number, and the shift register that receives the last input stream in the encoder provides the most significant bits in the state number.

In the outputs matrix, the element in the sth row and uth column denotes the encoder's output when the starting state is s-1 and the input bits have decimal representation u-1. To convert to decimal value, use the first output bit as the MSB.

Examples

These commands assemble the fields into a very simple trellis structure, and then verify the validity of the trellis structure.

```
trellis.numInputSymbols = 2;
trellis.numOutputSymbols = 2;
trellis.numStates = 2;
trellis.nextStates = [0 1;0 1];
trellis.outputs = [0 0;1 1];
[isok,status] = istrellis(trellis)
```

The output is below.

```
isok =

     1


status =
```

''

Another example of a trellis is in "Trellis Description of a Convolutional Code".

See Also

`poly2trellis` | `struct` | `convenc` | `vitdec`

How To

- "Convolutional Codes"

Purpose

Toggles random number generation mode for channel objects

Syntax

```
b = legacychannelsim
legacychannelsim(true)
legacychannelsim(false)
oldmode = legacychannelsim(newmode)
```

Description

`b = legacychannelsim` returns FALSE if the code you are running uses the R2009b (or later) version of the random number generator for `rayleighchan` or `ricianchan`. (By default, these use the 2009b random number generator.) It returns TRUE if pre-R2009b versions are used. See Version 4.4. (R2009b) Communications System Toolbox Release Notes for more information.

`legacychannelsim(true)` reverts the random number generation mode for channel objects to pre-2009b version.

Note `legacychannelsim(true)` will support the `reset(chan,randstate)` functionality.

`legacychannelsim(false)` sets the random number generation mode for channel objects to 2009b and later versions.

`oldmode = legacychannelsim(newmode)` sets the random number generation mode for channel objects to NEWMODE and returns the previous mode, OLDMODE.

lineareq

Purpose

Construct linear equalizer object

Syntax

```
eqobj = lineareq(nweights,alg)
eqobj = lineareq(nweights,alg,sigconst)
eqobj = lineareq(nweights,alg,sigconst,nsamp)
```

Description

The `lineareq` function creates an equalizer object that you can use with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see "Adaptive Algorithms".

`eqobj = lineareq(nweights,alg)` constructs a symbol-spaced linear equalizer object. The equalizer has `nweights` complex weights, which are initially all zeros. `alg` describes the adaptive algorithm that the equalizer uses; you should create `alg` using any of these functions: `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma`. The signal constellation of the desired output is `[-1 1]`, which corresponds to binary phase shift keying (BPSK).

`eqobj = lineareq(nweights,alg,sigconst)` specifies the signal constellation vector of the desired output.

`eqobj = lineareq(nweights,alg,sigconst,nsamp)` constructs a fractionally spaced linear equalizer object. The equalizer has `nweights` complex weights spaced at `T/nsamp`, where `T` is the symbol period and `nsamp` is a positive integer. `nsamp = 1` corresponds to a symbol-spaced equalizer.

Properties

The table below describes the properties of the linear equalizer object. To learn how to view or change the values of a linear equalizer object, see "Accessing Properties of an Equalizer".

Tip To initialize or reset the equalizer object `eqobj`, enter `reset(eqobj)`.

Property	Description
`EqType`	Fixed value, `'Linear Equalizer'`
`AlgType`	Name of the adaptive algorithm represented by `alg`
`nWeights`	Number of weights
`nSampPerSym`	Number of input samples per symbol (equivalent to `nsamp` input argument). This value relates to both the equalizer structure (see the use of K in “Fractionally Spaced Equalizers”) and an assumption about the signal to be equalized.
`RefTap` (except for CMA equalizers)	Reference tap index, between 1 and `nWeights`. Setting this to a value greater than 1 effectively delays the reference signal and the output signal by `RefTap-1` with respect to the equalizer's input signal.
`SigConst`	Signal constellation, a vector whose length is typically a power of 2
`Weights`	Vector of complex coefficients. This is the set of w_i values in the schematic in “Symbol-Spaced Equalizers”.
`WeightInputs`	Vector of tap weight inputs. This is the set of u_i values in the schematic in “Symbol-Spaced Equalizers”.

Property	Description
`ResetBeforeFiltering`	If `1`, each call to `equalize` resets the state of `eqobj` before equalizing. If `0`, the equalization process maintains continuity from one call to the next.
`NumSamplesProcessed`	Number of samples the equalizer processed since the last reset. When you create or reset `eqobj`, this property value is `0`.
Properties specific to the adaptive algorithm represented by `alg`	See reference page for the adaptive algorithm function that created `alg`: `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma`.

Relationships Among Properties

If you change `nWeights`, MATLAB maintains consistency in the equalizer object by adjusting the values of the properties listed below.

Property	Adjusted Value
`Weights`	`zeros(1,nWeights)`
`WeightInputs`	`zeros(1,nWeights)`
`StepSize` (Variable-step-size LMS equalizers)	`InitStep*ones(1,nWeights)`
`InvCorrMatrix` (RLS equalizers)	`InvCorrInit*eye(nWeights)`

An example illustrating relationships among properties is in "Linked Properties of an Equalizer Object".

Examples

For examples that use this function, see “Equalize Using a Training Sequence in MATLAB”, “Example: Equalizing Multiple Times, Varying the Mode”, and “Example: Adaptive Equalization Within a Loop”.

See Also

`lms` | `signlms` | `normlms` | `varlms` | `rls` | `cma` | `dfe` | `equalize`

How To

- “Equalization”

Purpose

Optimize quantization parameters using Lloyd algorithm

Syntax

```
[partition,codebook] = lloyds(training_set,initcodebook)
[partition,codebook] = lloyds(training_set,len)
[partition,codebook] = lloyds(training_set,...,tol)
[partition,codebook,distor] = lloyds(...)
[partition,codebook,distor,reldistor] = lloyds(...)
```

Description

`[partition,codebook] = lloyds(training_set,initcodebook)` optimizes the scalar quantization parameters `partition` and `codebook` for the training data in the vector `training_set`. `initcodebook`, a vector of length at least 2, is the initial guess of the codebook values. The output `codebook` is a vector of the same length as `initcodebook`. The output `partition` is a vector whose length is one less than the length of `codebook`.

See "Represent Partitions", "Represent Codebooks", or the reference page for `quantiz` in this chapter, for a description of the formats of `partition` and `codebook`.

Note `lloyds` optimizes for the data in `training_set`. For best results, `training_set` should be similar to the data that you plan to quantize.

`[partition,codebook] = lloyds(training_set,len)` is the same as the first syntax, except that the scalar argument `len` indicates the size of the vector `codebook`. This syntax does not include an initial codebook guess.

`[partition,codebook] = lloyds(training_set,...,tol)` is the same as the two syntaxes above, except that `tol` replaces 10^{-7} in condition 1 of the algorithm description below.

`[partition,codebook,distor] = lloyds(...)` returns the final mean square distortion in the variable `distor`.

`[partition,codebook,distor,reldistor] = lloyds(...)` returns a value `reldistor` that is related to the algorithm's termination. In condition 1 of the algorithm below, `reldistor` is the relative change in distortion between the last two iterations. In condition 2, `reldistor` is the same as `distor`.

Examples

The code below optimizes the quantization parameters for a sinusoidal transmission via a three-bit channel. Because the typical data is sinusoidal, `training_set` is a sampled sine wave. Because the channel can transmit three bits at a time, `lloyds` prepares a codebook of length 2^3.

```
% Generate a complete period of a sinusoidal signal.
x = sin([0:1000]*pi/500);
[partition,codebook] = lloyds(x,2^3)
```

The output is below.

```
partition =

  Columns 1 through 6

   -0.8540   -0.5973   -0.3017    0.0031    0.3077    0.6023

  Column 7

    0.8572


codebook =

  Columns 1 through 6

   -0.9504   -0.7330   -0.4519   -0.1481    0.1558    0.4575

  Columns 7 through 8

    0.7372    0.9515
```

Algorithms

`lloyds` uses an iterative process to try to minimize the mean square distortion. The optimization processing ends when either

- The relative change in distortion between iterations is less than 10^{-7}.
- The distortion is less than `eps*max(training_set)`, where `eps` is the MATLAB floating-point relative accuracy.

References

[1] Lloyd, S.P., "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, Vol. IT-28, March, 1982, pp. 129–137.

[2] Max, J., "Quantizing for Minimum Distortion," *IRE Transactions on Information Theory*, Vol. IT-6, March, 1960, pp. 7–12.

See Also

`quantiz` | `dpcmopt`

How To

- "Source Coding"

Purpose

Construct least mean square (LMS) adaptive algorithm object

Syntax

```
alg = lms(stepsize)
alg = lms(stepsize,leakagefactor)
```

Description

The `lms` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Adaptive Algorithms”.

`alg = lms(stepsize)` constructs an adaptive algorithm object based on the least mean square (LMS) algorithm with a step size of `stepsize`.

`alg = lms(stepsize,leakagefactor)` sets the leakage factor of the LMS algorithm. `leakagefactor` must be between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Properties

The table below describes the properties of the LMS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see “Access Properties of an Adaptive Algorithm”.

Property	Description
`AlgType`	Fixed value, `'LMS'`
`StepSize`	LMS step size parameter, a nonnegative real number
`LeakageFactor`	LMS leakage factor, a real number between 0 and 1

Examples

For examples that use this function, see “Equalize Using a Training Sequence in MATLAB”, “Example: Equalizing Multiple Times, Varying the Mode”, and “Example: Adaptive Equalization Within a Loop”.

Algorithms

Referring to the schematics presented in “Adaptive Algorithms”, define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of weights, w, this adaptive algorithm creates the new set of weights given by

(`LeakageFactor`) w + (`StepSize`) u^*e

where the * operator denotes the complex conjugate.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, John Wiley & Sons, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, NJ, Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, John Wiley & Sons, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

See Also

`signlms` | `normlms` | `varlms` | `rls` | `cma` | `lineareq` | `dfe` | `equalize`

How To

- “Equalization”

Purpose

Logarithm in Galois field

Syntax

```
y = log(x)
```

Description

`y = log(x)` computes the logarithm of each element in the Galois array `x`. `y` is an integer array that solves the equation `A.^y = x`, where `A` is the primitive element used to represent elements in `x`. More explicitly, the base `A` of the logarithm is `gf(2,x.m)` or `gf(2,x.m,x.prim_poly)`. All elements in `x` must be nonzero because the logarithm of zero is undefined.

Examples

The code below illustrates how the logarithm operation inverts exponentiation.

```
m = 4; x = gf([8 1 6; 3 5 7; 4 9 2],m);
y = log(x);
primel = gf(2,m); % Primitive element in the field
z = primel .^ y; % This is now the same as x.
ck = isequal(x,z)
```

The output is

```
ck =

     1
```

The code below shows that the logarithm of `1` is `0` and that the logarithm of the base (`primel`) is `1`.

```
m = 4; primel = gf(2,m);
yy = log([1, primel])
```

The output is

```
yy =

     0     1
```

Purpose

Generate root Zadoff-Chu sequence of complex symbols

Syntax

```
SEQ = lteZadoffChuSeq(R,N)
```

Description

`SEQ = lteZadoffChuSeq(R,N)` generates the `R`th root Zadoff-Chu sequence with length `N`, as defined in the LTE specifications [1]. The output SEQ is an `N`-length column vector of complex symbols.

The function generates the actual sequence using the following algorithm:

$$seq(m+1) = \exp(-j \cdot \pi \cdot R \cdot m \cdot (m+1) / N), \quad for\ m = 0, \ldots, N-1$$

This function uses a negative polarity on the argument of the exponent or a clockwise sequence of phases.

Input Arguments

R - Root of the Zadoff-Chu sequence
positive integer scalar

Example: `25`
Complex Number Support: Yes

N - Length of the Zadoff-Chu sequence.
positive integer scalar

Example: `139`
Complex Number Support: Yes

Output Arguments

SEQ - Zadoff-Chu output sequence
complex double-type column vector

The output sequence is a complex-valued column vector that contains the `R`th root Zadoff-Chu sequence of length `N`.

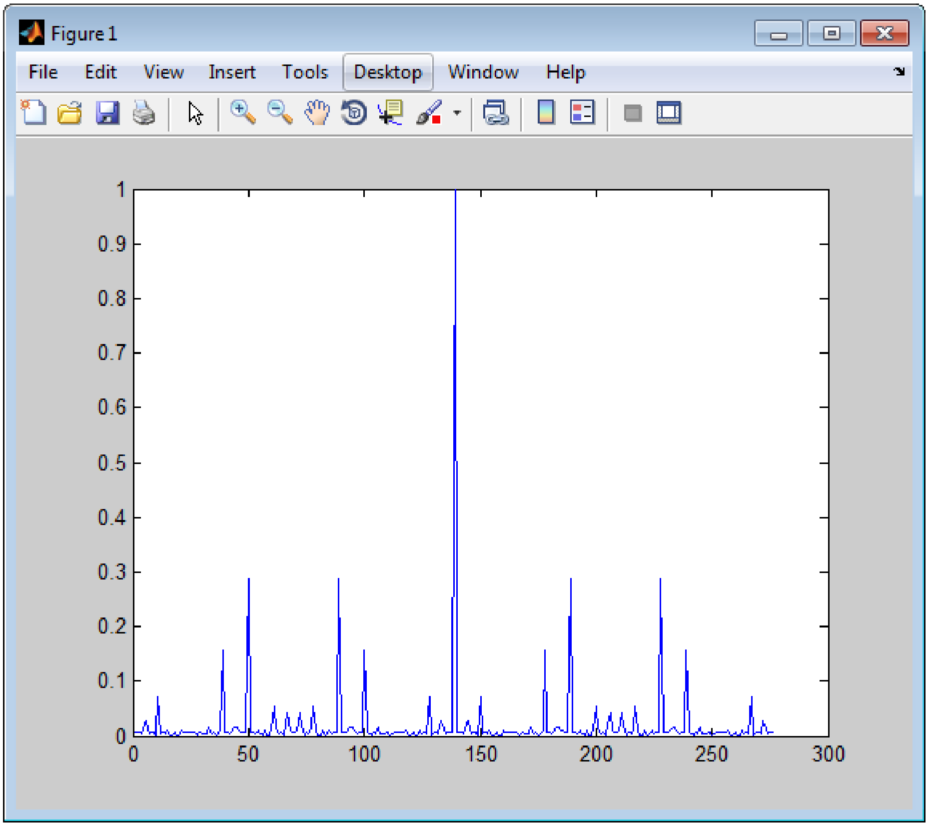
Examples

Examine the correlation properties of a Zadoff-Chu sequence

Generate the 25th root length-139 Zadoff-Chu sequence.

```
seq = lteZadoffChuSeq(25, 139);
plot(abs(xcorr(seq)./length(seq)))
```

MATLAB displays the following image:



References

[1] 3rd Generation Partnership Project: Technical Specification Group Radio Access Network. "Evolved Universal Terrestrial Radio Access (E-UTRA)," *Physical Channels and Modulation*, Release 10, 2010–2012, TS 36.211, Vol. 10.0.0.

See Also

`comm.GoldSequence` | `comm.PNSequence` |

Purpose

Generalized Marcum Q function

Syntax

```
Q = marcumq(a,b)
Q = marcumq(a,b,m)
```

Description

`Q = marcumq(a,b)` computes the Marcum Q function of `a` and `b`, defined by

$$Q(a,b) = \int_b^\infty x \exp\left(-\frac{x^2 + a^2}{2}\right) I_0(ax)dx$$

where `a` and `b` are nonnegative real numbers. In this expression, I_0 is the modified Bessel function of the first kind of zero order.

`Q = marcumq(a,b,m)` computes the generalized Marcum Q, defined by

$$Q\ (a,b) = \frac{1}{a^{m-1}} \int_b^\infty x^m \exp\left(-\frac{x^2 + a^2}{2}\right) I_{m-1}(ax)dx$$

where `a` and `b` are nonnegative real numbers, and `m` is a positive integer. In this expression, I_{m-1} is the modified Bessel function of the first kind of order m-1.

If any of the inputs is a scalar, it is expanded to the size of the other inputs.

References

[1] Cantrell, P. E., and A. K. Ojha, "Comparison of Generalized Q-Function Algorithms," *IEEE Transactions on Information Theory*, Vol. IT-33, July, 1987, pp. 591–596.

[2] Marcum, J. I., "A Statistical Theory of Target Detection by Pulsed Radar: Mathematical Appendix," RAND Corporation, Santa Monica, CA, Research Memorandum RM-753, July 1, 1948. Reprinted in *IRE Transactions on Information Theory*, Vol. IT-6, April, 1960, pp. 59–267.

[3] Shnidman, D. A., “The Calculation of the Probability of Detection and the Generalized Marcum Q-Function,” *IEEE Transactions on Information Theory*, Vol. IT-35, March, 1989, pp. 389–400.

See Also

`besseli`

Purpose

Convert mask vector to shift for shift register configuration

Syntax

```
shift = mask2shift(prpoly,mask)
```

Description

`shift = mask2shift(prpoly,mask)` returns the shift that is equivalent to a mask, for a linear feedback shift register whose connections are specified by the primitive polynomial `prpoly`. The `prpoly` input can have one of these formats:

- A binary vector that lists the coefficients of the primitive polynomial in order of descending powers
- An integer scalar whose binary representation gives the coefficients of the primitive polynomial, where the least significant bit is the constant term

The `mask` input is a binary vector whose length is the degree of the primitive polynomial.

Note To save time, `mask2shift` does not check that `prpoly` is primitive. If it is not primitive, the output is not meaningful. To find primitive polynomials, use `primpoly` or see [2].

For more information about how masks and shifts are related to pseudonoise sequence generators, see `shift2mask`.

Definition of Equivalent Shift

If A is a root of the primitive polynomial and m(A) is the mask polynomial evaluated at A, the equivalent shift s solves the equation A^s = m(A). To interpret the vector `mask` as a polynomial, treat `mask` as a list of coefficients in order of descending powers.

Examples

The first command below converts a mask of $x^3 + 1$ into an equivalent shift for the linear feedback shift register whose connections are specified by the primitive polynomial $x^4 + x^3 + 1$. The second command shows that a mask of 1 is equivalent to a shift of 0. In both cases, notice that the length of the `mask` vector is one less than the length of the `prpoly` vector.

```
s = mask2shift([1 1 0 0 1],[1 0 0 1])
s2 = mask2shift([1 1 0 0 1],[0 0 0 1])
```

The output is below.

```
s =

     4


s2 =

     0
```

References

[1] Lee, J. S., and L. E. Miller, *CDMA Systems Engineering Handbook*, Boston, Artech House, 1998.

[2] Simon, Marvin K., Jim K. Omura, et al., *Spread Spectrum Communications Handbook*, New York, McGraw-Hill, 1994.

See Also

`shift2mask` | `log` | `isprimitive` | `primpoly`

Purpose

Restore ordering of symbols by filling matrix by columns and emptying it by rows

Syntax

```
deintrlvd = matdeintrlv(data,Nrows,Ncols)
```

Description

`deintrlvd = matdeintrlv(data,Nrows,Ncols)` rearranges the elements in `data` by filling a temporary matrix with the elements column by column and then sending the matrix contents, row by row, to the output. `Nrows` and `Ncols` are the dimensions of the temporary matrix. If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

To use this function as an inverse of the `matintrlv` function, use the same `Nrows` and `Ncols` inputs in both functions. In that case, the two functions are inverses in the sense that applying `matintrlv` followed by `matdeintrlv` leaves data unchanged.

Examples

The code below illustrates the inverse relationship between `matintrlv` and `matdeintrlv`.

```
Nrows = 2; Ncols = 3;
data = [1 2 3 4 5 6; 2 4 6 8 10 12]';
a = matintrlv(data,Nrows,Ncols); % Interleave.
b = matdeintrlv(a,Nrows,Ncols) % Deinterleave.
```

The output below shows that `b` is the same as `data`.

```
b =

     1     2
     2     4
     3     6
     4     8
     5    10
     6    12
```

See Also

`matintrlv`

How To

- “Interleaving”

Purpose

Reorder symbols by filling matrix by rows and emptying it by columns

Syntax

```
intrlvd = matintrlv(data,Nrows,Ncols)
```

Description

`intrlvd = matintrlv(data,Nrows,Ncols)` rearranges the elements in `data` by filling a temporary matrix with the elements row by row and then sending the matrix contents, column by column, to the output. `Nrows` and `Ncols` are the dimensions of the temporary matrix. If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

Examples

The command below rearranges each of two columns of a matrix.

```
b = matintrlv([1 2 3 4 5 6; 2 4 6 8 10 12]',2,3)
b =

     1     2
     4     8
     2     4
     5    10
     3     6
     6    12
```

To form the first column of the output, the function creates the temporary 2-by-3 matrix `[1 2 3; 4 5 6]`. Then the function reads down each column of the temporary matrix to get `[1 4 2 5 3 6]`.

See Also

`matdeintrlv`

How To

- "Interleaving"

minpol

Purpose Find minimal polynomial of Galois field element

Syntax

```
pl = minpol(x)
```

Description `pl = minpol(x)` finds the minimal polynomial of each element in the Galois column vector, `x`. The output `pl` is an array in GF(2). The kth row of `pl` lists the coefficients, in order of descending powers, of the minimal polynomial of the kth element of `x`.

Note The output is in GF(2) even if the input is in a different Galois field.

Examples The code below uses `m = 4` and finds that the minimal polynomial of `gf(2,m)` is just the primitive polynomial used for the field GF(`2^m`). This is true for any value of `m`, not just the value used in the example.

```
m = 4;
A = gf(2,m)
pl = minpol(A)
```

The output is below. Notice that the row vector `[1 0 0 1 1]` represents the polynomial `D^4 + D + 1`.

```
A = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     2


pl = GF(2) array.

Array elements =

     1     0     0     1     1
```

Another example is in “Minimal Polynomials”.

See Also

`cosets`

How To

- “Polynomials over Galois Fields”

mldivide

Purpose Matrix left division \ of Galois arrays

Syntax

```
x = A\B
```

Description

x = A\B divides the Galois array A into B to produce a particular solution of the linear equation A*x = B. In the special case when A is a nonsingular square matrix, x is the unique solution, inv(A)*B, to the equation.

Examples

The code below shows that A \ eye(size(A)) is the inverse of the nonsingular square matrix A.

```
m = 4; A = gf([8 1 6; 3 5 7; 4 9 2],m);
Id = gf(eye(size(A)),m);
X = A \ Id;
ck1 = isequal(X*A, Id)
ck2 = isequal(A*X, Id)
```

The output is below.

```
ck1 =

     1


ck2 =

     1
```

Other examples are in “Solving Linear Equations”.

Limitations

The matrix A must be one of these types:

- A nonsingular square matrix
- A tall matrix such that A'*A is nonsingular
- A wide matrix such that A*A' is nonsingular

Algorithms

If `A` is an M-by-N tall matrix where M > N, `A \ B` is the same as `(A'*A) \ (A'*B)`.

If `A` is an M-by-N wide matrix where M < N, `A \ B` is the same as `A' * ((A*A') \ B)`. This solution is not unique.

How To

- "Linear Algebra in Galois Fields"

Purpose

Equalize linearly modulated signal using Viterbi algorithm

Syntax

```
y = mlseeq(x,chcffs,const,tblen,opmode)
y = mlseeq(x,chcffs,const,tblen,opmode,nsamp)
y = mlseeq(...,'rst',nsamp,preamble,postamble)
y = mlseeq(...,'cont',nsamp,...
init_metric,init_states,init_inputs)
[y,final_metric,final_states,final_inputs] = ...
mlseeq(...,'cont',...)
```

Description

`y = mlseeq(x,chcffs,const,tblen,`*`opmode`*`)` equalizes the baseband signal vector `x` using the Viterbi algorithm. `chcffs` is a vector that represents the channel coefficients. `const` is a complex vector that lists the points in the ideal signal constellation, in the same sequence that the system's modulator uses. `tblen` is the traceback depth. The equalizer traces back from the state with the best metric. *`opmode`* denotes the operation mode of the equalizer; the choices are described in the following table.

Value of opmode	Typical Usage
`'rst'`	Enables you to specify a preamble and postamble that accompany your data. The function processes `x` independently of data from any other invocations of this function. This mode incurs no output delay.
`'cont'`	Enables you to save the equalizer's internal state information for use in a subsequent invocation of this function. Repeated calls to this function are useful if your data is partitioned into a series of smaller vectors that you process within a loop, for example. This mode incurs an output delay of `tblen` symbols.

`y = mlseeq(x,chcffs,const,tblen,`*`opmode`*`,nsamp)` specifies the number of samples per symbol in `x`, that is, the oversampling factor.

The vector length of `x` must be a multiple of `nsamp`. When `nsamp` > 1, the `chcffs` input represents the oversampled channel coefficients.

Preamble and Postamble in Reset Operation Mode

`y = mlseeq(...,'rst',nsamp,preamble,postamble)` specifies the preamble and postamble that you expect to precede and follow, respectively, the data in the input signal. The vectors `preamble` and `postamble` consist of integers between 0 and M-1, where M is the order of the modulation, that is, the number of elements in `const`. To omit a preamble or postamble, specify `[]`.

When the function applies the Viterbi algorithm, it initializes state metrics in a way that depends on whether you specify a preamble and/or postamble:

- If the preamble is nonempty, the function decodes the preamble and assigns a metric of 0 to the decoded state. If the preamble does not decode to a unique state (that is, if the length of the preamble is less than the channel memory), the decoder assigns a metric of 0 to all states that can be represented by the preamble. The traceback path ends at one of the states represented by the preamble.
- If the preamble is unspecified or empty, the decoder initializes the metrics of all states to 0.
- If the postamble is nonempty, the traceback path begins at the smallest of all possible decoded states that are represented by the postamble.
- If the postamble is unspecified or empty, the traceback path starts at the state with the smallest metric.

Additional Syntaxes in Continuous Operation Mode

`y = mlseeq(...,'cont',nsamp,...`
`init_metric,init_states,init_inputs)` causes the equalizer to start with its state metrics, traceback states, and traceback inputs specified by `init_metric`, `init_states`, and `init_inputs`, respectively. These three inputs are typically the extra outputs from a previous call to this function, as in the syntax below. Each real number in `init_metric`

represents the starting state metric of the corresponding state. init_states and init_inputs jointly specify the initial traceback memory of the equalizer. The table below shows the valid dimensions and values of the last three inputs, where numStates is M^{L-1}, M is the order of the modulation, and L is the number of symbols in the channel's impulse response (with no oversampling). To use default values for all of the last three arguments, specify them as [],[],[].

Input Argument	Meaning	Matrix Size	Range of Values
init_metric	State metrics	1 row, numStates columns	Real numbers
init_states	Traceback states	numStates rows, tblen columns	Integers between 0 and numStates-1
init_inputs	Traceback inputs	numStates rows, tblen columns	Integers between 0 and M-1

[y,final_metric,final_states,final_inputs] = ... mlseeq(...,'cont',...) returns the normalized state metrics, traceback states, and traceback inputs, respectively, at the end of the traceback decoding process. final_metric is a vector with numStates elements that correspond to the final state metrics. final_states and final_inputs are both matrices of size numStates-by-tblen.

Examples

The example below illustrates how to use reset operation mode on an upsampled signal.

```
% Use 2-PAM.
M = 2; hMod = comm.PAMModulator(M); hDemod = comm.PAMDemodulator(M);
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)', ..
    'SNR',5);
const = step(hMod,(0:M-1)'); % PAM constellation
tblen =  10; % Traceback depth for equalizer
nsamp = 2; % Number of samples per symbol
```

```
msgIdx = randi([0 M-1],1000,1); % Random bits
msg = upsample(step(hMod,msgIdx),nsamp); % Modulated message
chcoeffs = [.986; .845; .237; .12345+.31i]; % Channel coefficients
chanest = chcoeffs; % Channel estimate
hMLSEE = comm.MLSEEqualizer('TracebackDepth',tblen,...
    'Channel',chanest, 'Constellation',const, 'SamplesPerSymbol', nsam
filtmsg = filter(chcoeffs,1,msg); % Introduce channel distortion.
msgRx = step(hChan,filtmsg); % Add Gaussian noise.
msgEq = step(hMLSEE,msgRx); % Equalize.
msgEqIdx = step(hDemod,msgEq); % Demodulate.
%Calculate BER
hErrorCalc = comm.ErrorRate;
berVec  = step(hErrorCalc, msgIdx, msgEqIdx);
ber  = berVec(1)
nerrs = berVec(2)
```

The output is shown below. Your results might vary because this example uses random numbers.

```
nerrs =

     1


ber =

    0.0010
```

The example in "Example: Continuous Operation Mode" illustrates how to use the final state and initial state arguments when invoking `mlseeq` repeatedly.

The example in "Use a Preamble in MATLAB" illustrates how to use a preamble.

References

[1] Proakis, John G., *Digital Communications*, Fourth Edition, New York, McGraw-Hill, 2001.

[2] Steele, Raymond, Ed., *Mobile Radio Communications*, Chichester, England, John Wiley & Sons, 1996.

See Also

`equalize`

How To

- "MLSE Equalizers"

Purpose

Scaling factor for normalizing modulation output

Syntax

```
scale = modnorm(const, 'avpow', avpow)
scale = modnorm(const, 'peakpow', peakpow)
```

Description

`scale = modnorm(const, 'avpow', avpow)` returns a scale factor for normalizing a PAM or QAM modulator output such that its average power is `avpow` (watts). `const` is a vector specifying the reference constellation used to generate the scale factor. The function assumes that the signal to be normalized has a minimum distance of 2.

`scale = modnorm(const, 'peakpow', peakpow)` returns a scale factor for normalizing a PAM or QAM modulator output such that its peak power is `peakpow` (watts).

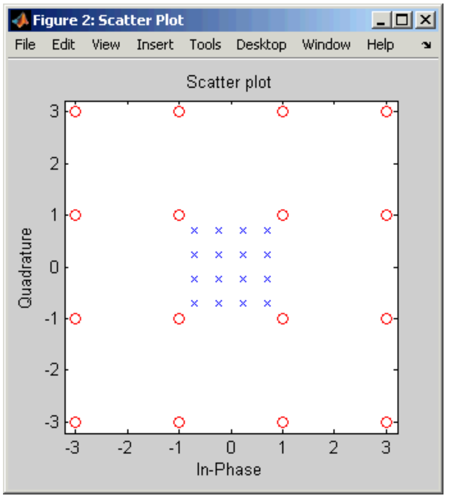
Examples

The code below illustrates how to use `modnorm` to transmit a quadrature amplitude modulated signal having a peak power of one watt.

```
M = 16; % Alphabet size
% QAM Modulation
hMod = comm.RectangularQAMModulator(M);
hDemod = comm.RectangularQAMDemodulator(M);
% AWGNChannel System object
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)'
    'SNR', 10);
const = step(hMod,(0:M-1)'); % Generate the constellation.
x = randi([0 M-1], 100,1);
scale = modnorm(const,'peakpow',1); % Compute scale factor.
y = scale * step(hMod,x); % Modulate and scale.
hChan.SignalPower = (y' * y)/ length(y); % Calculate Signal Power
ynoisy = step(hChan,y); % Transmit along noisy channel.

ynoisy_unscaled = ynoisy/scale; % Unscale at receiver end.
z = step(hDemod,ynoisy_unscaled); % Demodulate.

% See how scaling affects constellation.
h = scatterplot(const,1,0,'ro'); % Unscaled constellation
```

```
hold on; % Next plot will be in same figure window.
scatterplot(const*scale,1,0,'bx',h); % Scaled constellation
hold off;
```

In the plot below, the plotting symbol `o` marks points on the original QAM signal constellation, and the plotting symbol `x` marks points on the signal constellation as scaled by the output of the `modnorm` function. The channel in this example carries points from the scaled constellation.



Additional examples using `modnorm` are in "Examples of Signal Constellation Plots".

See Also

`pammod` | `pamdemod` | `qammod` | `qamdemod`

How To

- "Digital Modulation"

Purpose

Minimum shift keying demodulation

Syntax

```
z = mskdemod(y,nsamp)
z = mskdemod(y,nsamp,dataenc)
z = mskdemod(y,nsamp,dataenc,ini_phase)
z = mskdemod(y,nsamp,dataenc,ini_phase,ini_state)
[z,phaseout] = mskdemod(...)
[z,phaseout,stateout] = mskdemod(...)
```

Description

Warning

This function is obsolete and may be removed in the future. We strongly recommend that you use the `comm.MSKDemodulator` System object instead.

`z = mskdemod(y,nsamp)` demodulates the complex envelope `y` of a signal using the differentially encoded minimum shift keying (MSK) method. `nsamp` denotes the number of samples per symbol and must be a positive integer. The initial phase of the demodulator is 0. If `y` is a matrix with multiple rows and columns, the function treats the columns as independent channels and processes them independently.

`z = mskdemod(y,nsamp,`*`dataenc`*`)` specifies the method of encoding data for MSK. *`dataenc`* can be either `'diff'` for differentially encoded MSK or `'nondiff'` for nondifferentially encoded MSK.

`z = mskdemod(y,nsamp,`*`dataenc`*`,ini_phase)` specifies the initial phase of the demodulator. `ini_phase` is a row vector whose length is the number of channels in `y` and whose values are integer multiples of `pi/2`. To avoid overriding the default value of *`dataenc`*, set *`dataenc`* to `[]`.

`z = mskdemod(y,nsamp,`*`dataenc`*`,ini_phase,ini_state)` specifies the initial state of the demodulator. `ini_state` contains the last half symbol of the previously received signal. `ini_state` is an `nsamp`-by-C matrix, where C is the number of channels in `y`.

`[z,phaseout] = mskdemod(...)` returns the final phase of `y`, which is important for demodulating a future signal. The output `phaseout` has the same dimensions as the `ini_phase` input, and assumes the values 0, `pi/2`, `pi`, and `3*pi/2`.

`[z,phaseout,stateout] = mskdemod(...)` returns the final `nsamp` values of `y`, which is useful for demodulating the first symbol of a future signal. `stateout` has the same dimensions as the `ini_state` input.

Examples

The example below illustrates how to modulate and demodulate within a loop. To provide continuity from one iteration to the next, the syntaxes for `mskmod` and `mskdemod` use initial phases and/or state as both input and output arguments.

```
% Define parameters.
numbits = 99; % Number of bits per iteration
numchans = 2; % Number of channels (columns) in signal
nsamp = 16; % Number of samples per symbol

% Initialize.
numerrs = 0; % Number of bit errors seen so far
demod_ini_phase = zeros(1,numchans);  % Modulator phase
mod_ini_phase = zeros(1,numchans);    % Demodulator phase
ini_state = complex(zeros(nsamp,numchans)); % Demod. state

% Main loop
for iRuns = 1 : 10
    x = randi([0 1],numbits,numchans); % Binary signal
    [y,phaseout] = mskmod(x,nsamp,[],mod_ini_phase);
    mod_ini_phase = phaseout; % For next mskmod command
    [z, phaseout, stateout] = ...
       mskdemod(y,nsamp,[],demod_ini_phase,ini_state);
    ini_state = stateout;  % For next mskdemod command
    demod_ini_phase = phaseout; % For next mskdemod command
    numerrs = numerrs + biterr(x,z); % Cumulative bit errors
end
disp(['Total number of bit errors = ' num2str(numerrs)])
```

The output is as follows.

```
Total number of bit errors = 0
```

References

[1] Pasupathy, Subbarayan, "Minimum Shift Keying: A Spectrally Efficient Modulation," *IEEE Communications Magazine*, July, 1979, pp. 14–22.

See Also

`mskmod` | `fskmod` | `fskdemod`

How To

- "Digital Modulation"

mskmod

Purpose

Minimum shift keying modulation

Syntax

```
y = mskmod(x,nsamp)
y = mskmod(x,nsamp,dataenc)
y = mskmod(x,nsamp,dataenc,ini_phase)
[y,phaseout] = mskmod(...)
```

Description

Warning

This function is obsolete and may be removed in the future. We strongly recommend that you use the `comm.MSKModulator` System object instead.

`y = mskmod(x,nsamp)` outputs the complex envelope `y` of the modulation of the message signal `x` using differentially encoded minimum shift keying (MSK) modulation. The elements of `x` must be 0 or 1. `nsamp` denotes the number of samples per symbol in `y` and must be a positive integer. The initial phase of the MSK modulator is 0. If `x` is a matrix with multiple rows and columns, the function treats the columns as independent channels and processes them independently.

`y = mskmod(x,nsamp,`*`dataenc`*`)` specifies the method of encoding data for MSK. *`dataenc`* can be either `'diff'` for differentially encoded MSK or `'nondiff'` for nondifferentially encoded MSK.

`y = mskmod(x,nsamp,`*`dataenc`*`,ini_phase)` specifies the initial phase of the MSK modulator. `ini_phase` is a row vector whose length is the number of channels in `y` and whose values are integer multiples of `pi/2`. To avoid overriding the default value of *`dataenc`*, set *`dataenc`* to `[]`.

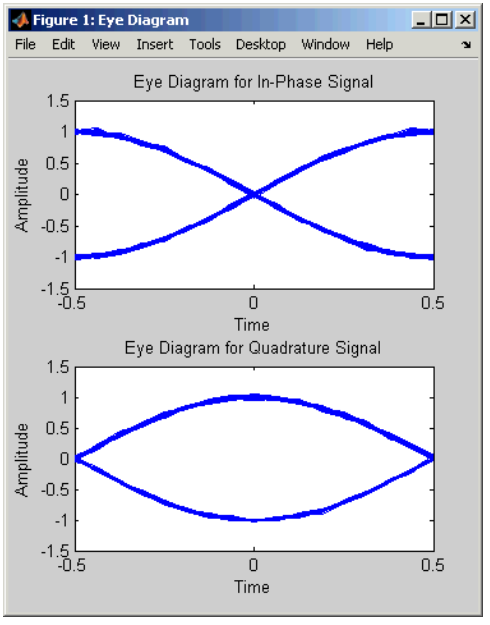
`[y,phaseout] = mskmod(...)` returns the final phase of `y`. This is useful for maintaining phase continuity when you are modulating a future bit stream with differentially encoded MSK. `phaseout` has the same dimensions as the `ini_phase` input, and assumes the values `0`, `pi/2`, `pi`, and `3*pi/2`.

Examples

Create an eye diagram from an MSK signal.

```
x = randi([0 1],99,1); % Random signal
```

```
y = mskmod(x,8,[],pi/2);
y = awgn(y,30,'measured');
eyediagram(y,16);
```



The example on the reference page for `mskdemod` also uses this function.

References

[1] Pasupathy, Subbarayan, "Minimum Shift Keying: A Spectrally Efficient Modulation," *IEEE Communications Magazine*, July, 1979, pp. 14–22.

See Also

`mskdemod` | `fskmod` | `fskdemod`

muxdeintrlv

Purpose

Restore ordering of symbols using specified shift registers

Syntax

```
deintrlved = muxdeintrlv(data,delay)
[deintrlved,state] = muxdeintrlv(data,delay)
[deintrlved,state] = muxdeintrlv(data,delay,init_state)
```

Description

`deintrlved = muxdeintrlv(data,delay)` restores the ordering of elements in `data` by using a set of internal shift registers, each with its own delay value. `delay` is a vector whose entries indicate how many symbols each shift register can hold. The length of `delay` is the number of shift registers. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`[deintrlved,state] = muxdeintrlv(data,delay)` returns a structure that holds the final state of the shift registers. `state.value` stores any unshifted symbols. `state.index` is the index of the next register to be shifted.

`[deintrlved,state] = muxdeintrlv(data,delay,init_state)` initializes the shift registers with the symbols contained in `init_state.value` and directs the first input symbol to the shift register referenced by `init_state.index`. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding interleaver.

Using an Interleaver-Deinterleaver Pair

To use this function as an inverse of the `muxintrlv` function, use the same `delay` input in both functions. In that case, the two functions are inverses in the sense that applying `muxintrlv` followed by `muxdeintrlv` leaves data unchanged, after you take their combined delay of `length(delay)*max(delay)` into account. To learn more about delays of convolutional interleavers, see “Delays of Convolutional Interleavers”.

Examples

The example below illustrates how to use the state input and output when invoking `muxdeintrlv` repeatedly. Notice that `[deintrlved1; deintrlved2]` is the same as `deintrlved`.

```
delay = [0 4 8 12]; % Delays in shift registers
symbols = 100; % Number of symbols to process
% Interleave random data.
intrlved = muxintrlv(randi([0 1],symbols,1),delay);

% Deinterleave some of the data, recording state for later use.
[deintrlved1,state] = muxdeintrlv(intrlved(1:symbols/2),delay);
% Deinterleave the rest of the data, using state as an input argument.
deintrlved2 = muxdeintrlv(intrlved(symbols/2+1:symbols),delay,state);

% Deinterleave all data in one step.
deintrlved = muxdeintrlv(intrlved,delay);

isequal(deintrlved,[deintrlved1; deintrlved2])
```

The output is below.

```
ans =

     1
```

Another example using this function is in "Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in MATLAB".

References

[1] Heegard, Chris, and Stephen B. Wicker, *Turbo Coding*, Boston, Kluwer Academic Publishers, 1999.

See Also

`muxintrlv`

How To

- "Interleaving"

muxintrlv

Purpose

Permute symbols using shift registers with specified delays

Syntax

```
intrlved = muxintrlv(data,delay)
[intrlved,state] = muxintrlv(data,delay)
[intrlved,state] = muxintrlv(data,delay,init_state)
```

Description

`intrlved = muxintrlv(data,delay)` permutes the elements in `data` by using internal shift registers, each with its own delay value. `delay` is a vector whose entries indicate how many symbols each shift register can hold. The length of `delay` is the number of shift registers. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`[intrlved,state] = muxintrlv(data,delay)` returns a structure that holds the final state of the shift registers. `state.value` stores any unshifted symbols. `state.index` is the index of the next register to be shifted.

`[intrlved,state] = muxintrlv(data,delay,init_state)` initializes the shift registers with the symbols contained in `init_state.value` and directs the first input symbol to the shift register referenced by `init_state.index`. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding deinterleaver.

Examples

The examples in “Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in MATLAB” and on the reference page for the `convintrlv` function use `muxintrlv`.

The example on the reference page for `muxdeintrlv` illustrates how to use the `state` output and `init_state` input with that function; the process is analogous for this function.

References

[1] Heegard, Chris, and Stephen B. Wicker, *Turbo Coding*, Boston, Kluwer Academic Publishers, 1999.

See Also

`muxdeintrlv` | `convintrlv` | `helintrlv`

How To

- “Interleaving”

Purpose Equivalent noise bandwidth of filter

Syntax

```
bw = noisebw(num, den, numsamp, Fs)
```

Description `bw = noisebw(num, den, numsamp, Fs)` returns the two-sided equivalent noise bandwidth, in Hz, of a digital lowpass filter given in descending powers of z by numerator vector `num` and denominator vector `den`. The bandwidth is calculated over `numsamp` samples of the impulse response. `Fs` is the sampling rate of the signal that the filter would process; this is used as a scaling factor to convert a normalized unitless quantity into a bandwidth in Hz.

Examples This example computes the equivalent noise bandwidth of a Butterworth filter over 100 samples of the impulse response.

```
Fs = 16; % Sampling rate
Fnyq = Fs/2; % Nyquist frequency
Fc = 0.5; % Carrier frequency
[num,den] = butter(2,Fc/Fnyq); % Butterworth filter
bw = noisebw(num,den,100,Fs)
```

The output is below.

```
bw =

    1.1049
```

Algorithms The two-sided equivalent noise bandwidth is

$$\frac{\mathrm{Fs}\sum_{i=1}^{N}\left|h(i)\right|^2}{\left|\sum_{i=1}^{N}h(i)\right|^2}$$

where h is the impulse response of the filter described by `num` and `den`, and N is `numsamp`.

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, New York, Plenum Press, 1992.

Purpose

Construct normalized least mean square (LMS) adaptive algorithm object

Syntax

```
alg = normlms(stepsize)
alg = normlms(stepsize,bias)
```

Description

The `normlms` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see "Adaptive Algorithms".

`alg = normlms(stepsize)` constructs an adaptive algorithm object based on the normalized least mean square (LMS) algorithm with a step size of `stepsize` and a bias parameter of zero.

`alg = normlms(stepsize,bias)` sets the bias parameter of the normalized LMS algorithm. `bias` must be between 0 and 1. The algorithm uses the bias parameter to overcome difficulties when the algorithm's input signal is small.

Properties

The table below describes the properties of the normalized LMS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see "Access Properties of an Adaptive Algorithm".

Property	Description
`AlgType`	Fixed value, `'Normalized LMS'`
`StepSize`	LMS step size parameter, a nonnegative real number

Property	Description
`LeakageFactor`	LMS leakage factor, a real number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, while a value of 0 corresponds to a memoryless update algorithm.
`Bias`	Normalized LMS bias parameter, a nonnegative real number

Examples

For an example that uses this function, see "Delays from Equalization".

Algorithms

Referring to the schematics presented in "Equalizer Structure", define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of weights, w, this adaptive algorithm creates the new set of weights given by

$$(\texttt{LeakageFactor})w + \frac{(\texttt{StepSize})u^{*}e}{u^{H}u + \texttt{Bias}}$$

where the * operator denotes the complex conjugate and H denotes the Hermitian transpose.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, John Wiley & Sons, 1998.

See Also

`lms` | `signlms` | `varlms` | `rls` | `cma` | `lineareq` | `dfe` | `equalize`

How To

- "Equalization"

oct2dec

Purpose

Convert octal to decimal numbers

Syntax

```
d = oct2dec(c)
```

Description

`d = oct2dec(c)` converts an octal matrix `c` to a decimal matrix `d`, element by element. In both octal and decimal representations, the rightmost digit is the least significant.

Examples

The command below converts a 2-by-2 octal matrix.

```
d = oct2dec([12 144;0 25])

d =

    10   100
     0    21
```

For instance, the octal number 144 is equivalent to the decimal number 100 because 144 (octal) = $1*8^2 + 4*8^1 + 4*8^0 = 64 + 32 + 4 = 100$.

See Also

`bi2de`

Purpose

Offset quadrature phase shift keying demodulation

Syntax

```
z = oqpskdemod(y)
z = oqpskdemod(y,ini_phase)
```

Description

`z = oqpskdemod(y)` demodulates the complex envelope `y` of an OQPSK modulated signal. The function implicitly downsamples by a factor of 2 because OQPSK does not permit an odd number of samples per symbol. If `y` is a matrix with multiple rows, the function processes the columns independently.

`z = oqpskdemod(y,ini_phase)` specifies the phase offset of the modulated signal in radians.

See Also

`oqpskmod` | `pskmod` | `pskdemod` | `qammod` | `qamdemod` | `modnorm`

How To

- "Digital Modulation"

oqpskmod

Purpose

Offset quadrature phase shift keying modulation

Syntax

```
y = oqpskmod(x)
y = oqpskmod(x,ini_phase)
```

Description

`y = oqpskmod(x)` outputs the complex envelope `y` of the modulation of the message signal `x` using offset quadrature phase shift keying (OQPSK) modulation. The message signal must consist of integers between 0 and 3. The function implicitly upsamples by a factor of 2 because OQPSK does not permit an odd number of samples per symbol. If `x` is a matrix with multiple rows, the function processes the columns independently.

`y = oqpskmod(x,ini_phase)` specifies the phase offset of the modulated signal in radians.

See Also

`oqpskdemod` | `pskmod` | `pskdemod` | `qammod` | `qamdemod` | `modnorm`

How To

- "Digital Modulation"

Purpose

Pulse amplitude demodulation

Syntax

```
z = pamdemod(y,M)
z = pamdemod(y,M,ini_phase)
z = pamdemod(y,M,ini_phase,symbol_order)
```

Description

`z = pamdemod(y,M)` demodulates the complex envelope `y` of a pulse amplitude modulated signal. `M` is the alphabet size. The ideal modulated signal should have a minimum Euclidean distance of 2.

`z = pamdemod(y,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

`z = pamdemod(y,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray-coded ordering.

Examples

The example in "Comparing Theoretical and Empirical Error Rates" uses this function.

See Also

`pammod` | `qamdemod` | `qammod` | `pskdemod` | `pskmod`

How To

- "Digital Modulation"

pammod

Purpose

Pulse amplitude modulation

Syntax

```
y = pammod(x,M)
y = pammod(x,M,ini_phase)
y = pammod(x,M,ini_phase,symbol_order)
```

Description

`y = pammod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using pulse amplitude modulation. `M` is the alphabet size. The message signal must consist of integers between 0 and `M-1`. The modulated signal has a minimum Euclidean distance of 2. If `x` is a matrix with multiple rows, the function processes the columns independently.

`y = pammod(x,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

`y = pammod(x,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray constellation ordering.

Examples

The example in “Comparing Theoretical and Empirical Error Rates” uses this function.

See Also

`pamdemod` | `qammod` | `qamdemod` | `pskmod` | `pskdemod`

How To

- “Digital Modulation”

Purpose

Plot channel characteristics with channel visualization tool

Syntax

```
plot(h)
```

Description

`plot(h)`, where `h` is a channel object, launches the channel visualization tool. This GUI tool allows you to plot channel characteristics in various ways. See “Channel Visualization” for details.

Examples

Examples using this plotting tool are found in “Examples of Using the Channel Visualization Tool”.

See Also

`filter` | `rayleighchan` | `ricianchan`

Purpose

Phase demodulation

Syntax

```
z = pmdemod(y,Fc,Fs,phasedev)
z = pmdemod(y,Fc,Fs,phasedev,ini_phase)
```

Description

`z = pmdemod(y,Fc,Fs,phasedev)` demodulates the phase-modulated signal `y` at the carrier frequency `Fc` (hertz). `z` and the carrier signal have sampling rate `Fs` (hertz), where `Fs` must be at least `2*Fc`. The `phasedev` argument is the phase deviation of the modulated signal, in radians.

`z = pmdemod(y,Fc,Fs,phasedev,ini_phase)` specifies the initial phase of the modulated signal, in radians.

Examples

The example in "Analog Modulation with Additive White Gaussian Noise (AWGN) Using MATLAB" uses `pmdemod`.

See Also

`pmmod` | `fmmod` | `fmdemod`

How To

- "Digital Modulation"

Purpose

Phase modulation

Syntax

```
y = pmmod(x,Fc,Fs,phasedev)
y = pmmod(x,Fc,Fs,phasedev,ini_phase)
```

Description

`y = pmmod(x,Fc,Fs,phasedev)` modulates the message signal `x` using phase modulation. The carrier signal has frequency `Fc` (hertz) and sampling rate `Fs` (hertz), where `Fs` must be at least `2*Fc`. The `phasedev` argument is the phase deviation of the modulated signal in radians.

`y = pmmod(x,Fc,Fs,phasedev,ini_phase)` specifies the initial phase of the modulated signal in radians.

Examples

The example in “Analog Modulation with Additive White Gaussian Noise (AWGN) Using MATLAB” uses `pmmod`.

See Also

`pmdemod` | `fmmod` | `fmdemod`

How To

- “Digital Modulation”

Purpose

Convert convolutional code polynomials to trellis description

Syntax

```
trellis = poly2trellis(ConstraintLength,CodeGenerator)
trellis = poly2trellis(ConstraintLength,CodeGenerator,...
 FeedbackConnection)
```

Description

The `poly2trellis` function accepts a polynomial description of a convolutional encoder and returns the corresponding trellis structure description. The output of `poly2trellis` is suitable as an input to the `convenc` and `vitdec` functions, and as a mask parameter for the Convolutional Encoder, Viterbi Decoder, and APP Decoder blocks in Communications System Toolbox software.

`trellis = poly2trellis(ConstraintLength,CodeGenerator)` performs the conversion for a rate k/n feedforward encoder. `ConstraintLength` is a 1-by-k vector that specifies the delay for the encoder's k input bit streams. `CodeGenerator` is a k-by-n matrix of octal numbers that specifies the n output connections for each of the encoder's k input bit streams.

`trellis = poly2trellis(ConstraintLength,CodeGenerator,... FeedbackConnection)` is the same as the syntax above, except that it applies to a feedback, not feedforward, encoder. `FeedbackConnection` is a 1-by-k vector of octal numbers that specifies the feedback connections for the encoder's k input bit streams.

For both syntaxes, the output is a MATLAB structure whose fields are as in the table below.

Fields of the Output Structure trellis for a Rate k/n Code

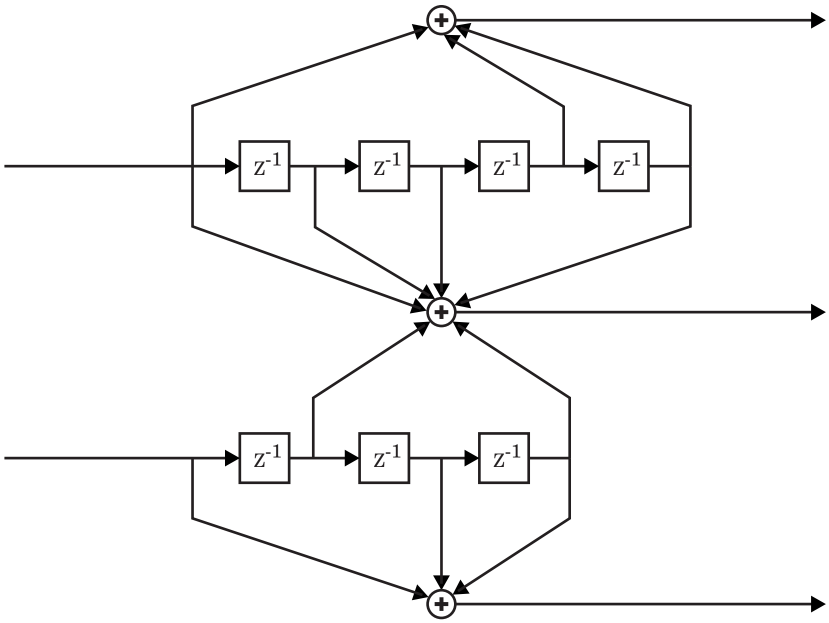
Field in trellis Structure	Dimensions	Meaning
`numInputSymbols`	Scalar	Number of input symbols to the encoder: 2^k
`numOutputSymbols`	Scalar	Number of output symbols from the encoder: 2^n
`numStates`	Scalar	Number of states in the encoder
`nextStates`	`numStates`-by-2^k matrix	Next states for all combinations of current state and current input
`outputs`	`numStates`-by-2^k matrix	Outputs (in octal) for all combinations of current state and current input

For more about this structure, see the reference page for the `istrellis` function.

Examples

An example of a rate 1/2 encoder is in "Polynomial Description of a Convolutional Code".

As another example, consider the rate 2/3 feedforward convolutional encoder depicted in the figure below. The reference page for the `convenc` function includes an example that uses this encoder.



For this encoder, the `ConstraintLength` vector is [5,4] and the `CodeGenerator` matrix is [23,35,0; 0,5,13]. The output below reveals part of the corresponding trellis structure description of this encoder.

```
trellis = poly2trellis([5 4],[23 35 0; 0 5 13])

trellis =

     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 128
          nextStates: [128x4 double]
             outputs: [128x4 double]
```

The scalar field `trellis.numInputSymbols` has the value 4 because the combination of two input bit streams can produce four different input symbols. Similarly, `trellis.numOutputSymbols` is 8 because the three output bit streams can produce eight different output symbols.

The scalar field `trellis.numStates` is 128 (that is, 2^7) because each of the encoder's seven memory registers can have one of two binary values.

To get details about the matrix fields `trellis.nextStates` and `trellis.outputs`, inquire specifically about them. As an example, the command below displays the first five rows of the 128-by-4 matrix `trellis.nextStates`.

```
trellis.nextStates(1:5,:)

ans =

     0    64     8    72
     0    64     8    72
     1    65     9    73
     1    65     9    73
     2    66    10    74
```

This first row indicates that if the encoder starts in the zeroth state and receives input bits of 00, 01, 10, or 11, respectively, the next state will be the 0th, 64th, 8th, or 72nd state, respectively. The 64th state means that the bottom-left memory register in the diagram contains the value 1, while the other six memory registers contain zeros.

See Also

`istrellis` | `convenc` | `vitdec`

How To

- "Convolutional Codes"

primpoly

Purpose

Find primitive polynomials for Galois field

Syntax

```
pr = primpoly(m)
pr = primpoly(m,opt)
pr = primpoly(m...,'nodisplay')
```

Description

`pr = primpoly(m)` returns the primitive polynomial for GF(`2^m`), where `m` is an integer between `2` and `16`. The Command Window displays the polynomial using "`D`" as an indeterminate quantity. The output argument `pr` is an integer whose binary representation indicates the coefficients of the polynomial.

`pr = primpoly(m,`*`opt`*`)` returns one or more primitive polynomials for GF(`2^m`). The output `pol` depends on the argument *`opt`* as shown in the table below. Each element of the output argument `pr` is an integer whose binary representation indicates the coefficients of the corresponding polynomial. If no primitive polynomial satisfies the constraints, `pr` is empty.

opt	Meaning of pr
`'min'`	One primitive polynomial for GF(`2^m`) having the smallest possible number of nonzero terms
`'max'`	One primitive polynomial for GF(`2^m`) having the greatest possible number of nonzero terms
`'all'`	All primitive polynomials for GF(`2^m`)
Positive integer k	All primitive polynomials for GF(`2^m`) that have k nonzero terms

`pr = primpoly(m...,'nodisplay')` prevents the function from displaying the result as polynomials in "`D`" in the Command Window. The output argument `pr` is unaffected by the `'nodisplay'` option.

Examples

The first example below illustrates the formats that `primpoly` uses in the Command Window and in the output argument `pr`. The subsequent examples illustrate the display options and the use of the *`opt`* argument.

```
pr = primpoly(4)

pr1 = primpoly(5,'max','nodisplay')

pr2 = primpoly(5,'min')

pr3 = primpoly(5,2)

pr4 = primpoly(5,3);
```

The output is below.

```
Primitive polynomial(s) =

D^4+D^1+1

pr =

    19



pr1 =

    61



Primitive polynomial(s) =
```

```
D^5+D^2+1

pr2 =

    37


No primitive polynomial satisfies the given constraints.

pr3 =

     []


Primitive polynomial(s) =

D^5+D^2+1
D^5+D^3+1
```

See Also

`isprimitive`

How To

- "Galois Field Computations"

Purpose

Phase shift keying demodulation

Syntax

```
z = pskdemod(y,M)
z = pskdemod(y,M,ini_phase)
z = pskdemod(y,M,ini_phase,symbol_order)
```

Description

`z = pskdemod(y,M)` demodulates the complex envelope `y` of a PSK modulated signal. `M` is the alphabet size and must be an integer power of 2. The initial phase of the modulation is zero. If `y` is a matrix with multiple rows and columns, the function processes the columns independently.

`z = pskdemod(y,M,ini_phase)` specifies the initial phase of the modulation in radians.

`z = pskdemod(y,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray-coded ordering.

Examples

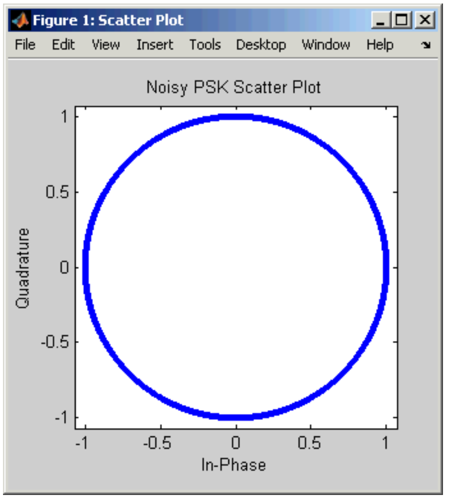
The example below compares PSK and PAM (phase amplitude modulation) to show that PSK is more sensitive to phase noise. This is the expected result because the PSK constellation is circular, and the PAM constellation is linear.

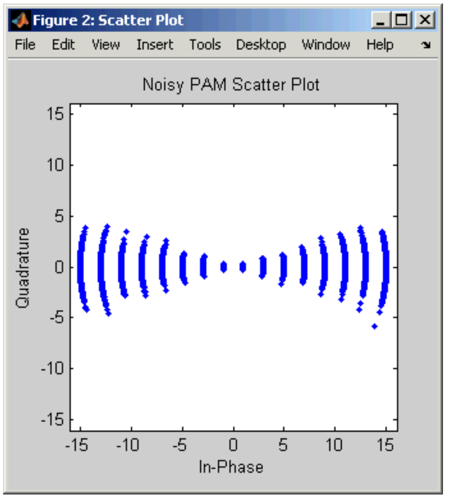
```
len = 10000; % Number of symbols
M = 16; % Size of alphabet
msg = randi([0 M-1],len,1); % Original signal

% Modulate using both PSK and PAM,
% to compare the two methods.
txpsk = pskmod(msg,M);
txpam = pammod(msg,M);

% Perturb the phase of the modulated signals.
phasenoise = randn(len,1)*.015;
rxpsk = txpsk.*exp(j*2*pi*phasenoise);
```

```
rxpam = txpam.*exp(j*2*pi*phasenoise);

% Create a scatter plot of the received signals.
scatterplot(rxpsk); title('Noisy PSK Scatter Plot')
scatterplot(rxpam); title('Noisy PAM Scatter Plot')

% Demodulate the received signals.
recovpsk = pskdemod(rxpsk,M);
recovpam = pamdemod(rxpam,M);

% Compute number of symbol errors in each case.
numerrs_psk = symerr(msg,recovpsk)
numerrs_pam = symerr(msg,recovpam)
```

The output and scatter plots are below. Your results might vary because this example uses random numbers.

```
numerrs_psk =

   374


numerrs_pam =

     1
```





See Also

`pskmod` | `qamdemod` | `qammod` | `dpskmod` | `dpskdemod` | `modnorm`

How To

- "Digital Modulation"

Purpose

Phase shift keying modulation

Syntax

```
y = pskmod(x,M)
y = pskmod(x,M,ini_phase)
y = pskmod(x,M,ini_phase,symbol_order)
```

Description

`y = pskmod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using phase shift keying modulation. `M` is the alphabet size and must be an integer power of 2. The message signal must consist of integers between 0 and `M-1`. The initial phase of the modulation is zero. If `x` is a matrix with multiple rows and columns, the function processes the columns independently.

`y = pskmod(x,M,ini_phase)` specifies the initial phase of the modulation in radians.

`y = pskmod(x,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray constellation ordering.

Examples

The examples in “Create 16-PSK Constellation Scatter Plot” and on the reference page for `pskdemod` use this function.

See Also

`dpskmod` | `dpskdemod` | `pskdemod` | `pammod` | `pamdemod` | `qammod` | `qamdemod` | `modnorm`

How To

- “Digital Modulation”

Purpose

Quadrature amplitude demodulation

Syntax

```
z = qamdemod(y,M)
z = qamdemod(y,M,ini_phase)
z = qamdemod(y,M,ini_phase,symbol_order)
```

Description

`z = qamdemod(y,M)` demodulates the complex envelope `y` of a quadrature amplitude modulated signal. `M` is the alphabet size and must be an integer power of 2. The constellation is the same as in `qammod`. If `y` is a matrix with multiple rows, the function processes the columns independently.

`z = qamdemod(y,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

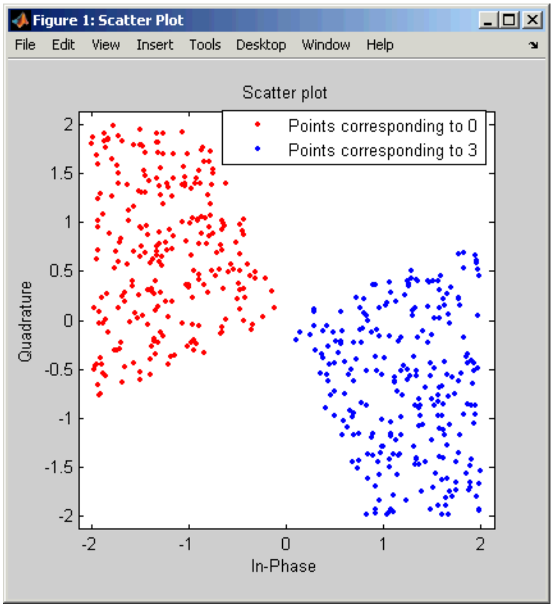
`z = qamdemod(y,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray-coded ordering.

Examples

The code below suggests which regions in the complex plane are associated with different digits that can form the output of the demodulator. The code demodulates random points, looks for points that were demapped to the digits 0 and 3, and plots those points in red and blue, respectively. Notice that the regions reflect a rotation of the signal constellation by `pi/8`.

```
% Construct [in-phase, quadrature] for random points.
y = 4*(rand(1000,1)-1/2)+j*4*(rand(1000,1)-1/2);
% Demodulate using an initial phase of pi/8.
z = qamdemod(y,4,pi/8);
% Find indices of points that mapped to the digits 0 and 3.
red = find(z==0);
blue = find(z==3);
% Plot points corresponding to 0 and 3.
h = scatterplot(y(red,:),1,0,'r.'); hold on
scatterplot(y(blue,:),1,0,'b.',h);
```

```
legend('Points corresponding to 0','Points corresponding to 3');
hold off
```



Another example using this function is in “Compute the Symbol Error Rate”.

See Also

`qammod` | `genqamdemod` | `genqammod` | `pamdemod` | `modnorm`

How To

- “Digital Modulation”

Purpose Quadrature amplitude modulation

Syntax

```
y = qammod(x,M)
y = qammod(x,M,ini_phase)
y = qammod(x,M,ini_phase,symbol_order)
```

Description

`y = qammod(x,M)` outputs the complex envelope `y` of the modulation of the message signal `x` using quadrature amplitude modulation. `M` is the alphabet size and must be an integer power of 2. The message signal must consist of integers between 0 and `M-1`. The signal constellation is rectangular or cross-shaped, and the nearest pair of points in the constellation is separated by 2. If `x` is a matrix with multiple rows, the function processes the columns independently.

`y = qammod(x,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

`y = qammod(x,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray constellation ordering.

Examples Examples using this function are in "Compute the Symbol Error Rate" and "Examples of Signal Constellation Plots".

See Also `qamdemod` | `genqammod` | `genqamdemod` | `pammod` | `pamdemod` | `modnorm`

How To

- "Digital Modulation"

qfunc

Purpose Q function

Syntax

```
y = qfunc(x)
```

Description

`y = qfunc(x)` is one minus the cumulative distribution function of the standardized normal random variable, evaluated at each element of the real array `x`. For a scalar `x`, the formula is

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} \exp(-t^2/2)dt$$

The Q function is related to the complementary error function, erfc, according to

$$Q(x) = \frac{1}{2} \operatorname{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

Examples

The example below computes the Q function on a matrix, element by element.

```
x = [0 1 2; 3 4 5];
format short e % Switch to floating point format for displays.
y = qfunc(x)
format % Return to default format for displays.
```

The output is below.

```
y =

  5.0000e-001  1.5866e-001  2.2750e-002
  1.3499e-003  3.1671e-005  2.8665e-007
```

See Also

`qfuncinv` | `erf` | `erfc` | `erfcx` | `erfinv` | `erfcinv`

Purpose

Inverse Q function

Syntax

```
y = qfuncinv(x)
```

Description

`y = qfuncinv(x)` returns the argument of the Q function at which the Q function's value is `x`. The input `x` must be a real array with elements between 0 and 1, inclusive.

For a scalar `x`, the Q function is one minus the cumulative distribution function of the standardized normal random variable, evaluated at `x`. The Q function is defined as

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty \exp(-t^2/2)dt$$

The Q function is related to the complementary error function, erfc, according to

$$Q(x) = \frac{1}{2}\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

Examples

The example below illustrates the inverse relationship between `qfunc` and `qfuncinv`.

```
x1 = [0 1 2; 3 4 5];
y1 = qfuncinv(qfunc(x1)) % Invert qfunc to recover x1.
x2 = 0:.2:1;
y2 = qfunc(qfuncinv(x2)) % Invert qfuncinv to recover x2.
```

The output is below.

```
y1 =

     0     1     2
     3     4     5
```

```
y2 =

         0    0.2000    0.4000    0.6000    0.8000    1.0000
```

See Also

qfunc | erf | erfc | erfcx | erfinv | erfcinv

Purpose

Produce quantization index and quantized output value

Syntax

```
index = quantiz(sig,partition)
[index,quants] = quantiz(sig,partition,codebook)
[index,quants,distor] = quantiz(sig,partition,codebook)
```

Description

`index = quantiz(sig,partition)` returns the quantization levels in the real vector signal `sig` using the parameter `partition`. `partition` is a real vector whose entries are in strictly ascending order. If `partition` has length n, `index` is a vector whose kth entry is

- 0 if `sig`(k) ≤ `partition`(1)
- m if `partition`(m) < `sig`(k) ≤ `partition`(m+1)
- n if `partition`(n) < `sig`(k)

`[index,quants] = quantiz(sig,partition,codebook)` is the same as the syntax above, except that `codebook` prescribes a value for each partition in the quantization and `quants` contains the quantization of `sig` based on the quantization levels and prescribed values. `codebook` is a vector whose length exceeds the length of `partition` by one. `quants` is a row vector whose length is the same as the length of `sig`. `quants` is related to `codebook` and `index` by

```
quants(ii) = codebook(index(ii)+1);
```

where `ii` is an integer between 1 and `length(sig)`.

`[index,quants,distor] = quantiz(sig,partition,codebook)` is the same as the syntax above, except that `distor` estimates the mean square distortion of this quantization data set.

Examples

The command below rounds several numbers between 1 and 100 up to the nearest multiple of 10. `quants` contains the rounded numbers, and `index` tells which quantization level each number is in.

```
[index,quants] = quantiz([3 34 84 40 23],10:10:90,10:10:100)
```

The output is below.

```
index =

     0     3     8     3     2


quants =

    10    40    90    40    30
```

See Also

lloyds | dpcmenco | dpcmdeco

How To

- "Quantize a Signal"

Purpose

Restore ordering of symbols using random permutation

Syntax

```
deintrlvd = randdeintrlv(data,state)
```

Description

`deintrlvd = randdeintrlv(data,state)` restores the original ordering of the elements in `data` by inverting a random permutation. The `state` parameter initializes the random number generator that the function uses to determine the permutation. `state` is either a scalar or a 35x1 vector, and is described in the `rand` function, which is used in `randintrlv`. The function is predictable for a given state, but different states produce different permutations. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

To use this function as an inverse of the `randintrlv` function, use the same `state` input in both functions. In that case, the two functions are inverses in the sense that applying `randintrlv` followed by `randdeintrlv` leaves data unchanged.

This function uses, by default, the Mersenne Twister algorithm by Nishimura and Matsumoto.

Note Using the `state` parameter causes this function to switch random generators to use the `'state'` algorithm of the `rand` function.

See `rand` for details on the generator algorithm.

Examples

For an example using random interleaving and deinterleaving, see "Improve Error Rate Using Block Interleaving in MATLAB".

See Also

`rand` | `randintrlv`

How To

- "Interleaving"

randerr

Purpose Generate bit error patterns

Syntax

```
out = randerr(m)
out = randerr(m,n)
out = randerr(m,n,errors)
out = randerr(m,n,prob,state)
out = randerr(m,n,prob,s)
```

Description For all syntaxes, `randerr` treats each row of `out` independently.

`out = randerr(m)` generates an `m`-by-`m` binary matrix, each row of which has exactly one nonzero entry in a random position. Each allowable configuration has an equal probability.

`out = randerr(m,n)` generates an `m`-by-`n` binary matrix, each row of which has exactly one nonzero entry in a random position. Each allowable configuration has an equal probability.

`out = randerr(m,n,errors)` generates an `m`-by-`n` binary matrix, where `errors` determines how many nonzero entries are in each row:

- If `errors` is a scalar, it is the number of nonzero entries in each row.
- If `errors` is a row vector, it lists the possible number of nonzero entries in each row.
- If `errors` is a matrix having two rows, the first row lists the possible number of nonzero entries in each row and the second row lists the probabilities that correspond to the possible error counts.

Once `randerr` determines the *number* of nonzero entries in a given row, each configuration of that number of nonzero entries has equal probability.

`out = randerr(m,n,prob,state)` is the same as the syntax above, except that it first resets the state of the uniform random number generator `rand` to the integer `state`.

Note This usage is deprecated and may be removed in a future release. Instead of `state`, use `s`, as in the following example.

This function uses, by default, the Mersenne Twister algorithm by Nishimura and Matsumoto.

Note Using the `state` parameter causes this function to switch random generators to use the `'state'` algorithm of the `rand` function.

See `rand` for details on the generator algorithm.

`out = randerr(m,n,prob,s)` causes rand to use the random stream s. See RandStream for more details.

Examples

The examples below generate an 8-by-7 binary matrix, each row of which is equally likely to have either zero or two nonzero entries, and then alter the scenario by making it three times as likely that a row has two nonzero entries. Notice in the latter example that the second row of the `error` parameter sums to one.

```
out = randerr(8,7,[0 2])

out2 = randerr(8,7,[0 2; .25 .75])
```

Sample output is below.

```
out =

     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
     0     0     1     0     0     0     1
     1     0     1     0     0     0     0
     0     0     0     0     0     0     0
```

```
     0     0     0     0     0     0     0
     0     0     0     0     1     1     0
     1     0     1     0     0     0     0


out2 =

     0     0     0     0     0     0     0
     1     0     0     0     0     0     1
     1     0     0     0     0     0     1
     0     0     0     1     0     1     0
     0     0     0     0     0     0     0
     0     1     0     0     0     0     1
     0     0     0     0     0     0     0
     1     0     0     0     1     0     0
```

See Also

`rand` | `randsrc` | `randint`

How To

- "Sources and Sinks"

Purpose

Generate matrix of uniformly distributed random integers

Note MathWorks will remove the `randint` function from a future version of the Communications System Toolbox software. We strongly recommend that you use `randi` instead.

Syntax

```
out = randint
out = randint(m)
out = randint(m,n)
out = randint(m,n,rg)
out = randint(m,n,rg,state)
```

Description

`out = randint` generates a random scalar that is either 0 or 1, with equal probability.

`out = randint(m)` generates an `m`-by-`m` binary matrix, each of whose entries independently takes the value 0 with probability 1/2.

`out = randint(m,n)` generates an `m`-by-`n` binary matrix, each of whose entries independently takes the value 0 with probability 1/2.

`out = randint(m,n,rg)` generates an `m`-by-`n` integer matrix. If `rg` is zero, `out` is a zero matrix. Otherwise, the entries are uniformly distributed and independently chosen from the range

- [0, `rg`-1] if `rg` is a positive integer
- [`rg`+1, 0] if `rg` is a negative integer
- Between `min` and `max`, inclusive, if `rg = [min,max]` or `[max,min]`

`out = randint(m,n,rg,state)` is the same as the syntax above, except that it first resets the state of the uniform random number generator `rand` to the integer `state`.

This function uses, by default, the Mersenne Twister algorithm by Nishimura and Matsumoto.

Note Using the `state` parameter causes this function to switch random generators to use the `'state'` algorithm of the `rand` function.

See `rand` for details on the generator algorithm.

Examples

To generate a 10-by-10 matrix whose elements are uniformly distributed in the range from 0 to 7, use either of the following commands.

```
out = randint(10,10,[0,7]);
```

```
out = randint(10,10,8);
```

See Also

`rand` | `randsrc` | `randerr`

How To

- "Sources and Sinks"

Purpose

Reorder symbols using random permutation

Syntax

```
intrlvd = randintrlv(data,state)
```

Description

`intrlvd = randintrlv(data,state)` rearranges the elements in `data` using a random permutation. The `state` parameter initializes the random number generator that the function uses to determine the permutation. `state` is either a scalar or a 35x1 vector, and is described in the `rand` function, which is used in `randintrlv`. The function is predictable and invertible for a given state, but different states produce different permutations. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

This function uses, by default, the Mersenne Twister algorithm by Nishimura and Matsumoto.

Note Using the `state` parameter causes this function to switch random generators to use the `'state'` algorithm of the`rand` function.

See `rand` for details on the generator algorithm.

Examples

For an example using random interleaving and deinterleaving, see “Improve Error Rate Using Block Interleaving in MATLAB”.

See Also

`rand` | `randdeintrlv`

How To

- “Interleaving”

randseed

Purpose

Generate prime numbers for use as random number seeds

Syntax

```
out = randseed
out = randseed(state)
out = randseed(state,m)
out = randseed(state,m,n)
out = randseed(state,m,n,rmin)
out = randseed(state,m,n,rmin,rmax)
```

Description

The `randseed` function produces random prime numbers that work well as seeds for random source blocks or noisy channel blocks in Communications System Toolbox software. It is recommended you use the `randseed` function when specifying the initial seed parameters of the following blocks: Gaussian, Rayleigh, and Rician Noise Generator.

Note The `randseed` function uses a local stream of numbers that is independent from the global stream of numbers in the MATLAB software. Use of this function does not affect the state of the global random number stream.

`out = randseed` generates a random prime number between 31 and 2^{17}-1, using the MATLAB function `rand`.

`out = randseed(state)` generates a random prime number after setting the state of `rand` to the positive integer `state`. This syntax produces the same output for a particular value of `state`.

`out = randseed(state,m)` generates a column vector of `m` random primes.

`out = randseed(state,m,n)` generates an `m`-by-`n` matrix of random primes.

`out = randseed(state,m,n,rmin)` generates an `m`-by-`n` matrix of random primes between `rmin` and 2^{17}-1.

out = randseed(state,m,n,rmin,rmax) generates an m-by-n matrix of random primes between rmin and rmax.

Examples

To generate a two-element sample-based row vector of random bits using the Bernoulli Random Binary Generator block, you can set **Probability of a zero** to [0.1 0.5] and set **Initial seed** to randseed(391,1,2).

To generate three streams of random data from three different blocks in a single model, you can define out = randseed(93,3) in the MATLAB workspace and then set the three blocks' **Initial seed** parameters to out(1), out(2), and out(3), respectively.

See Also

rand | primes

Purpose

Generate random matrix using prescribed alphabet

Syntax

```
out = randsrc
out = randsrc(m)
out = randsrc(m,n)
out = randsrc(m,n,alphabet)
out = randsrc(m,n,[alphabet; prob])
out = randsrc(m,n,...,state);
out = randsrc(m,n,...,s);
```

Description

`out = randsrc` generates a random scalar that is either -1 or 1, with equal probability.

`out = randsrc(m)` generates an `m`-by-`m` matrix, each of whose entries independently takes the value -1 with probability 1/2, and 1 with probability 1/2.

`out = randsrc(m,n)` generates an `m`-by-`n` matrix, each of whose entries independently takes the value -1 with probability 1/2, and 1 with probability 1/2.

`out = randsrc(m,n,alphabet)` generates an `m`-by-`n` matrix, each of whose entries is independently chosen from the entries in the row vector `alphabet`. Each entry in `alphabet` occurs in `out` with equal probability. Duplicate values in `alphabet` are ignored.

`out = randsrc(m,n,[alphabet; prob])` generates an `m`-by-`n` matrix, each of whose entries is independently chosen from the entries in the row vector `alphabet`. Duplicate values in `alphabet` are ignored. The row vector `prob` lists corresponding probabilities, so that the symbol `alphabet(k)` occurs with probability `prob(k)`, where `k` is any integer between one and the number of columns of `alphabet`. The elements of `prob` must add up to 1.

`out = randsrc(m,n,...,state);` is the same as the two preceding syntaxes, except that it first resets the state of the uniform random number generator `rand` to the integer `state`.

Note This usage is deprecated and may be removed in a future release. Instead of `state`, use `s`, as in the following example.

This function uses, by default, the Mersenne Twister algorithm by Nishimura and Matsumoto.

Note Using the `state` parameter causes this function to switch random generators to use the `'state'` algorithm of the `rand` function.

See `rand` for details on the generator algorithm.

`out = randsrc(m,n,...,s);` causes rand to use the random stream s. See RandStream for more details.

Examples

To generate a 10-by-10 matrix whose elements are uniformly distributed among members of the set {-3,-1,1,3}, you can use either of these commands.

```
out = randsrc(10,10,[-3 -1 1 3]);
```

```
out = randsrc(10,10,[-3 -1 1 3; .25 .25 .25 .25]);
```

To skew the probability distribution so that -1 and 1 each occur with probability .3, while -3 and 3 each occur with probability .2, use this command.

```
out = randsrc(10,10,[-3 -1 1 3; .2 .3 .3 .2]);
```

See Also

`rand` | `randint` | `randerr`

Purpose

Construct Rayleigh fading channel object

Syntax

```
chan = rayleighchan(ts,fd)
chan = rayleighchan(ts,fd,tau,pdb)
chan = rayleighchan
```

Description

`chan = rayleighchan(ts,fd)` constructs a frequency-flat ("single path") Rayleigh fading channel object. `ts` is the sample time of the input signal, in seconds. `fd` is the maximum Doppler shift, in hertz. You can model the effect of the channel on a signal `x` by using the syntax `y = filter(chan,x)`.

`chan = rayleighchan(ts,fd,tau,pdb)` constructs a frequency-selective ("multiple path") fading channel object that models each discrete path as an independent Rayleigh fading process. `tau` is a vector of path delays, each specified in seconds. `pdb` is a vector of average path gains, each specified in dB.

With the above two syntaxes, a smaller `fd` (a few hertz to a fraction of a hertz) leads to slower variations, and a larger `fd` (a couple hundred hertz) to faster variations.

`chan = rayleighchan` constructs a frequency-flat Rayleigh channel object with no Doppler shift. This is a static channel. The sample time of the input signal is irrelevant for frequency-flat static channels.

Properties

The tables below describe the properties of the channel object, `chan`, that you can set and that MATLAB technical computing software sets automatically. To learn how to view or change the values of a channel object, see "Display Object Properties" or "Change Object Properties".

Writeable Properties

Property	Description
InputSamplePeriod	Sample period of the signal on which the channel acts, measured in seconds.
DopplerSpectrum	Doppler spectrum object(s). The default is a Jakes Doppler object.
MaxDopplerShift	Maximum Doppler shift of the channel, in hertz (applies to all paths of a channel).
PathDelays	Vector listing the delays of the discrete paths, in seconds.
AvgPathGaindB	Vector listing the average gain of the discrete paths, in decibels.
NormalizePathGains	If 1, the Rayleigh fading process is normalized such that the expected value of the path gains' total power is 1.
StoreHistory	If this value is 1, channel state information needed by the channel visualization tool is stored as the channel filter function processes the signal. The default value is 0.

Writeable Properties (Continued)

Property	Description
`StorePathGains`	If set to `1`, the complex path gain vector is stored as the channel filter function processes the signal. The default value is `0`.
`ResetBeforeFiltering`	If `1`, each call to `filter` resets the state of `chan` before filtering. If `0`, the fading process maintains continuity from one call to the next.

Read-Only Properties

Property	Description	When MATLAB Sets or Updates Value
`ChannelType`	Fixed value, `'Rayleigh'`	When you create object
`PathGains`	Complex vector listing the current gains of the discrete paths. When you create or reset `chan`, `PathGains` is a random vector influenced by `AvgPathGaindB` and `NormalizePathGains`.	When you create object, reset object, or use it to filter a signal

Read-Only Properties (Continued)

Property	Description	When MATLAB Sets or Updates Value
`ChannelFilterDelay`	Delay of the channel filter, measured in samples. The ChannelFilterDelay property returns a delay value that is valid only if the first value of the PathGain is the biggest path gain. In other words, main channel energy is in the first path.	When you create object or change ratio of `InputSamplePeriod` to `PathDelays`
`NumSamplesProcessed`	Number of samples the channel processed since the last reset. When you create or reset `chan`, this property value is `0`.	When you create object, reset object, or use it to filter a signal

Relationships Among Properties

The `PathDelays` and `AvgPathGaindB` properties of the channel object must always have the same vector length, because this length equals the number of discrete paths of the channel. The `DopplerSpectrum` property must either be a single Doppler object or a vector of Doppler objects with the same length as `PathDelays`.

If you change the length of `PathDelays`, MATLAB truncates or zero-pads the value of `AvgPathGaindB` if necessary to adjust its vector length (MATLAB may also change the values of read-only properties such as `PathGains` and `ChannelFilterDelay`). If `DopplerSpectrum` is a vector of Doppler objects, and you increase or decrease the length of `PathDelays`, MATLAB will add Jakes Doppler objects or remove elements from `DopplerSpectrum`, respectively, to make it the same length as `PathDelays`.

If `StoreHistory` is set to `1` (the default is `0`), the object stores channel state information as the channel filter function processes the signal. You can then visualize this state information through a GUI using the `plot (channel)` method.

Note Setting `StoreHistory` to `1` will result in a slower simulation. If you do not want to visualize channel state information using `plot (channel)`, but want to access the complex path gains, then set `StorePathGains` to `1`, while keeping `StoreHistory` as `0`.

Visualization of Channel

The characteristics of a channel can be plotted using the channel visualization tool. You can use the channel visualization tool in Normal mode and Accelerator mode. For more information, see "Channel Visualization".

Examples

Several examples using this function are in "Fading Channels".

The example below illustrates that when you change the value of `PathDelays`, MATLAB automatically changes the values of other properties to make their vector lengths consistent with that of the new value of `PathDelays`.

```
c1 = rayleighchan(1e-5,130) % Create object.
c1.PathDelays = [0 1e-6]    % Change the number of delays.
% MATLAB automatically changes the size of c1.AvgPathGaindB,
% c1.PathGains, and c1.ChannelFilterDelay.
```

The output below displays all the properties of the channel object before and after the change in the value of the `PathDelays` property. In the second listing of properties, the `AvgPathGaindB`, `PathGains`, and `ChannelFilterDelay` properties all have different values compared to the first listing of properties.

```
c1 =

            ChannelType: 'Rayleigh'
      InputSamplePeriod: 1.0000e-005
        DopplerSpectrum: [1x1 doppler.jakes]
        MaxDopplerShift: 130
             PathDelays: 0
          AvgPathGaindB: 0
     NormalizePathGains: 1
           StoreHistory: 0
              PathGains: 0.2035 + 0.1014i
     ChannelFilterDelay: 0
   ResetBeforeFiltering: 1
    NumSamplesProcessed: 0


c1 =

            ChannelType: 'Rayleigh'
      InputSamplePeriod: 1.0000e-005
        DopplerSpectrum: [1x1 doppler.jakes]
        MaxDopplerShift: 130
             PathDelays: [0 1.0000e-006]
          AvgPathGaindB: [0 0]
     NormalizePathGains: 1
           StoreHistory: 0
              PathGains: [0.6108 - 0.4688i 0.1639 - 0.0027i]
     ChannelFilterDelay: 4
   ResetBeforeFiltering: 1
    NumSamplesProcessed: 0
```

Algorithms

The methodology used to simulate fading channels is described in “Methodology for Simulating Multipath Fading Channels:”. The properties of the channel object are related to the quantities of the latter section as follows:

- The `InputSamplePeriod` property contains the value of T_s.

- The `PathDelays` vector property contains the values of $\{\tau_k\}$, where $1 \le k \le K$.
- The `PathGains` read-only property contains the values of $\{a_k\}$, where $1 \le k \le K$.
- The `AvgPathGaindB` vector property contains the values of $10\log_{10}\left\{E\left[\left|a_k\right|^2\right]\right\}$, where $1 \le k \le K$, and $E[\cdot]$ denotes statistical expectation.
- The `ChannelFilterDelay` read-only property contains the value of N_1.

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

See Also

`ricianchan` | `filter` | `plot (channel)` | `reset`

How To

- “Fading Channels”

Purpose

Design raised cosine finite impulse response (FIR) filter

Note MathWorks will remove the `rcosfir` function from a future version of the Communications System Toolbox software. While the product still supports this function, you should use `fdesign.pulseshaping` instead.

Syntax

```
b = rcosfir(R,n_T,rate,T)
b = rcosfir(R,n_T,rate,T,filter_type)
rcosfir(...)
rcosfir(...,colr)
[b,sample_time] = rcosfir(...)
```

Optional Inputs

Input	Default Value
n_T	3
rate	5
T	1

Description

The `rcosfir` function designs the same filters that the `rcosine` function designs when the latter's `type_flag` argument includes `'fir'`. However, `rcosine` is somewhat easier to use.

The time response of the raised cosine filter has the form

$$h(t) = \frac{\sin(\pi t/T)}{(\pi t/T)} \cdot \frac{\cos(\pi R t/T)}{(1-4R^2t^2/T^2)}$$

`b = rcosfir(R,n_T,rate,T)` designs a raised cosine filter and returns a vector `b` of length`(n_T(2) - n_T(1))*rate + 1`. The filter's rolloff factor is `R`, a real number between 0 and 1, inclusive. `T` is the duration of each bit in seconds. n_T is a scalar or a vector of length 2. If n_T is specified as a scalar, the filter length is 2*n_T+1 input samples. If n_T is a vector, it specifies the extent of the filter. In this case, the filter length is n_T(2)-n_T(1)+1 input samples (or (n_T(2)-n_T(1))*rate+1 output samples).

rate is the number of points in each input symbol period of length T. rate must be greater than 1. The input sample rate is T samples per second, while the output sample rate is T*rate samples per second.

The order of the FIR filter is

```
(n_T(2)-n_T(1))*rate
```

The arguments n_T, rate, and T are optional inputs whose default values are 3, 5, and 1, respectively.

b = rcosfir(R,n_T,rate,T,*filter_type*) designs a square-root raised cosine filter if *filter_type* is 'sqrt'. If *filter_type* is 'normal', this syntax is the same as the previous one.

The impulse response of a square root raised cosine filter is

$$h(t) = 4R\frac{\cos((1+R)\pi t/T) + \dfrac{\sin((1-R)\pi t/T)}{4R\dfrac{t}{T}}}{\pi\sqrt{T}(1-(4Rt/T)^2)}$$

rcosfir(...) produces plots of the time and frequency responses of the raised cosine filter.

rcosfir(...,colr) uses the string colr to determine the plotting color. The choices for colr are the same as those listed for the plot function.

[b,sample_time] = rcosfir(...) returns the FIR filter and its sample time.

Examples

The commands below compare different rolloff factors.

```
rcosfir(0);
subplot(211); hold on;
subplot(212); hold on;
rcosfir(.5,[],[],[],[],'r-');
rcosfir(1,[],[],[],[],'g-');
```

References

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

See Also

`rcosiir` | `rcosflt` | `rcosine` | `firrcos`

How To

- rcosdemo
- “Filtering”

Purpose

Filter input signal using raised cosine filter

Syntax

```
y = rcosflt(x,Fd,Fs)
y = rcosflt(x,Fd,Fs,'type_flag',r,delay,tol)
y = rcosflt(x,Fd,Fs,'filter_type/Fs',r,delay,tol)
y = rcosflt(x,Fd,Fs,'filter_type/filter',num,den)
y = rcosflt(x,Fd,Fs,'filter_type/filter',num,den,delay)
y = rcosflt(x,Fd,Fs,'filter_type/filter/Fs',num,den...)
[y,t] = rcosflt(...)
```

Note MathWorks will remove the `rcosflt` function from a future version of the Communications System Toolbox software. We strongly recommend that you use `fdesign.pulseshaping` instead.

Optional Inputs

Input	Default Value
filter_type	fir/normal
r	0.5
delay	3
tol	0.01
den	1

Description

The function `rcosflt` passes an input signal through a raised cosine filter. You can either let `rcosflt` design a raised cosine filter automatically or you can specify the raised cosine filter yourself using input arguments.

Designing the Filter Automatically

`y = rcosflt(x,Fd,Fs)` designs a raised cosine FIR filter and then filters the input signal `x` using it. The sample frequency for the digital input signal `x` is `Fd`, and the sample frequency for the output signal `y` is Fs. The ratio Fs/Fd must be an integer. In the course of filtering, rcosflt upsamples the data by a factor of Fs/Fd, by inserting zeros between samples. The order of the filter is 1+2*delay*Fs/Fd, where delay is 3 by default. If x is a vector, then the sizes of x and y are related by this equation.

```
length(y) = (length(x) + 2 * delay)*Fs/Fd
```

Otherwise, y is a matrix, each of whose columns is the result of filtering the corresponding column of x.

y = rcosflt(x,Fd,Fs,'*type_flag*',r,delay,tol) designs a raised cosine FIR or IIR filter and then filters the input signal x using it. The ratio Fs/Fd must be an integer. r is the rolloff factor for the filter, a real number in the range [0, 1]. delay is the filter's group delay, measured in input samples. The actual group delay in the filter design is delay/Fd seconds. The input tol is the tolerance in the IIR filter design. FIR filter design does not use tol.

The characteristics of x, Fd, Fs, and y are as in the first syntax.

The fourth input argument, '*type_flag*', determines the type of filter that rcosflt should design and can have up to three components: filter type, sample frequency, and filter.

Values of filter_type to Determine the Type of Filter

Type of Filter	Value of filter_type
FIR raised cosine filter	fir or fir/normal
IIR raised cosine filter	iir or iir/normal
Square-root FIR raised cosine filter	fir/sqrt
Square-root IIR raised cosine filter	iir/sqrt

y = rcosflt(x,Fd,Fs,'*filter_type*/Fs',r,delay,tol) is the same as the previous syntax, except that it assumes that x has sample frequency Fs. This syntax does not upsample x any further. If x is a vector, then the relative sizes of x and y are related by this equation.

```
length(y) = length(x) + (2 * delay * Fs/Fd)
```

As before, if x is a nonvector matrix, y is a matrix, each of whose columns is the result of filtering the corresponding column of x.

Specifying the Filter Using Input Arguments

y = rcosflt(x,Fd,Fs,'*filter_type*/filter',num,den) filters the input signal x using a filter whose transfer function numerator and denominator are given in num and den, respectively. If *type_filter* includes fir, then omit den. This syntax uses the same arguments x, Fd, Fs, and *type_filter* as explained in the first and second syntaxes above.

y = rcosflt(x,Fd,Fs,'*filter_type*/filter',num,den,delay) uses delay in the same way that the rcosine function uses it. This syntax assumes that the filter described by num, den, and delay was designed using rcosine.

As before, if x is a nonvector matrix, y is a matrix each of whose columns is the result of filtering the corresponding column of x.

y = rcosflt(x,Fd,Fs,'*filter_type*/filter/Fs',num,den...) is the same as the earlier syntaxes, except that it assumes that x has sample frequency Fs instead of Fd. This syntax does not upsample x any further. If x is a vector, the relative sizes of x and y are related by this equation.

```
length(y) = length(x) + (2 * delay * Fs/Fd)
```

Additional Output

[y,t] = rcosflt(...) outputs t, a vector that contains the sampling time points of y.

References

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

See Also

`rcosine` | `rcosfir` | `rcosiir`

How To

- rcosdemo
- “Filtering”

Purpose

Design raised cosine infinite impulse response (IIR) filter

Syntax

```
[num,den] = rcosiir(R,T_delay,rate,T,tol)
[num,den] = rcosiir(R,T_delay,rate,T,tol,type_filter)
rcosiir(...)
rcosiir(...,colr)
[num,den,sample_time] = rcosiir(...)
```

Note MathWorks will remove the `rcosiir` function from a future version of the Communications System Toolbox software.

Optional Inputs

Input	Default Value
`T_delay`	3
`rate`	5
`T`	1
`tol`	0.01

Description

The `rcosiir` function designs the same filters that the `rcosine` function designs when the latter's `type_flag` argument includes `'iir'`. However, `rcosine` is somewhat easier to use.

The time response of the raised cosine filter has the form

$$h(t) = \frac{\sin(\pi t / T)}{(\pi t / T)} \cdot \frac{\cos(\pi R t / T)}{(1 - 4R^2 t^2 / T^2)}$$

`[num,den] = rcosiir(R,T_delay,rate,T,tol)` designs an IIR approximation of an FIR raised cosine filter, and returns the numerator and denominator of the IIR filter. The filter's rolloff factor is `R`, a real number between 0 and 1, inclusive. `T` is the symbol period in seconds.

The filter's group delay is T_delay symbol periods. rate is the number of sample points in each interval of duration T. rate must be greater than 1. The input sample rate is T samples per second, while the output sample rate is T*rate samples per second. If tol is an integer greater than 1, it becomes the order of the IIR filter; if tol is less than 1, it indicates the relative tolerance for rcosiir to use when selecting the order based on the singular values.

The arguments T_delay, rate, T, and tol are optional inputs whose default values are 3, 5, 1, and 0.01, respectively.

[num,den] = rcosiir(R,T_delay,rate,T,tol,*type_filter*) designs a square-root raised cosine filter if *type_filter* is 'sqrt'. If *type_filter* is 'normal', this syntax is the same as the previous one.

rcosiir(...) plots the time and frequency responses of the raised cosine filter.

rcosiir(...,colr) uses the string colr to determine the plotting color. The choices for colr are the same as those listed for the plot function.

[num,den,sample_time] = rcosiir(...) returns the transfer function and the sample time of the IIR filter.

Examples

The script below compares different values of T_delay.

```
rcosiir(0,10);
subplot(211); hold on;
subplot(212); hold on;
col = ['r-';'g-';'b-';'m-';'c-';'w-'];
R = [8,6,4,3,2,1];
for ii = R
   rcosiir(0,ii,[],[],[],[],col(find(R==ii),:));
end;
```

This example shows how the filter's frequency response more closely approximates that of the ideal raised cosine filter as T_delay increases.

References

[1] Kailath, Thomas, *Linear Systems*, Englewood Cliffs, N.J., Prentice-Hall, 1980.

[2] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

See Also

`rcosfir` | `rcosflt` | `rcosine`

How To

- rcosdemo
- "Filtering"

Purpose

Design raised cosine filter

Syntax

```
num = rcosine(Fd,Fs)
[num,den] = rcosine(Fd,Fs,type_flag)
[num,den] = rcosine(Fd,Fs,type_flag,r)
[num,den] = rcosine(Fd,Fs,type_flag,r,delay)
[num,den] = rcosine(Fd,Fs,type_flag,r,delay,tol)
```

Note MathWorks will remove the `rcosine` function from a future version of the Communications System Toolbox software. We strongly recommend that you use `fdesign.pulseshaping` instead.

Description

`num = rcosine(Fd,Fs)` designs a finite impulse response (FIR) raised cosine filter and returns its transfer function. The digital input signal has sampling frequency `Fd`. The sampling frequency for the filter is `Fs`. The ratio `Fs/Fd` must be a positive integer greater than 1. The default rolloff factor is .5. The filter's group delay, which is the time between the input to the filter and the filter's peak response, is three input samples. Equivalently, the group delay is 3/`Fd` seconds.

`[num,den] = rcosine(Fd,Fs,type_flag)` designs a raised cosine filter using directions in the string variable *`type_flag`*. Filter types are listed in the table below, along with the corresponding values of *`type_flag`*.

Types of Filter and Corresponding Values of type_flag

Type of Filter	Value of type_flag
Finite impulse response (FIR)	`'default'` or `'fir/normal'`
Infinite impulse response (IIR)	`'iir'` or ''`iir/normal`''
Square-root raised cosine FIR	`'sqrt'` or `'fir/sqrt'`
Square-root raised cosine IIR	`'iir/sqrt'`

The default tolerance value in IIR filter design is 0.01.

`[num,den] = rcosine(Fd,Fs,`*`type_flag`*`,r)` specifies the rolloff factor, `r`. The rolloff factor is a real number in the range [0, 1].

`[num,den] = rcosine(Fd,Fs,`*`type_flag`*`,r,delay)` specifies the filter's group delay, measured in input samples. `delay` is a positive integer. The actual group delay in the filter design is `delay/Fd` seconds.

`[num,den] = rcosine(Fd,Fs,`*`type_flag`*`,r,delay,tol)` specifies the tolerance in the IIR filter design. FIR filter design does not use `tol`.

References

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

See Also

`rcosflt | rcosiir | rcosfir`

How To

- rcosdemo
- "Filtering"

Purpose

Rectangular pulse shaping

Syntax

```
y = rectpulse(x,nsamp)
```

Description

`y = rectpulse(x,nsamp)` applies rectangular pulse shaping to `x` to produce an output signal having `nsamp` samples per symbol. Rectangular pulse shaping means that each symbol from `x` is repeated `nsamp` times to form the output `y`. If `x` is a matrix with multiple rows, the function treats each column as a channel and processes the columns independently.

Note To insert zeros between successive samples of `x` instead of repeating the samples of `x`, use the `upsample` function instead.

Examples

An example in "Combine Pulse Shaping and Filtering with Modulation" uses this function in conjunction with modulation.

The code below processes two independent channels, each containing three symbols of data. In the pulse-shaped matrix `y`, each symbol contains four samples.

```
nsamp = 4; % Number of samples per symbol
nsymb = 3; % Number of symbols
s = RandStream('mt19937ar', 'Seed', 0);
ch1 = randi(s, [0 1], nsymb, 1); % Random binary channel
ch2 = [1:nsymb]';
x = [ch1 ch2] % Two-channel signal
y = rectpulse(x,nsamp)
```

The output is below. In `y`, each column corresponds to one channel and each row corresponds to one sample. Also, the first four rows of `y` correspond to the first symbol, the next four rows of `y` correspond to the second symbol, and the last four rows of `y` correspond to the last symbol.

```
x =
```

```
     1     1
     1     2
     0     3


y =

     1     1
     1     1
     1     1
     1     1
     1     2
     1     2
     1     2
     1     2
     0     3
     0     3
     0     3
     0     3
```

See Also

intdump | upsample | rcosflt

Purpose

Reset channel object

Syntax

```
reset(chan)
reset(chan,randstate)
```

Description

`reset(chan)` resets the channel object `chan`, initializing the `PathGains` and `NumSamplesProcessed` properties as well as internal filter states. This syntax is useful when you want the effect of creating a new channel.

`reset(chan,randstate)` resets the channel object `chan` and initializes the state of the random number generator that the channel uses. `randstate` is a two-element column vector. This syntax is useful when you want to repeat previous numerical results that started from a particular state.

Note `reset(chan,randstate)` will not support `randstate` in a future release. See the `legacychannelsim` function for more information.

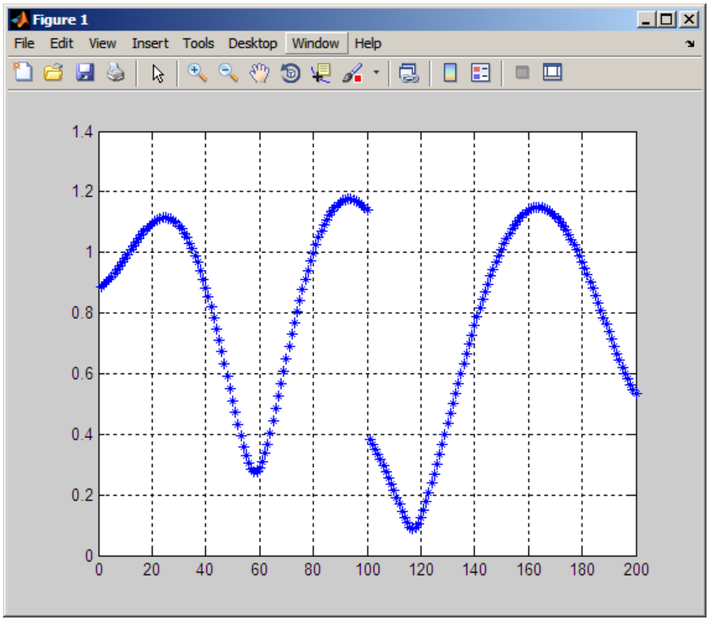
Examples

The example below shows how to obtain repeatable results. The example chooses a state for the random number generator immediately after defining the channel object and later resets the random number generator to that state.

```
% Set up channel.
% Assume you want to maintain continuity
% from one filtering operation to the next, except
% when you explicitly reset the channel.
c = rayleighchan(1e-4,100);
c.ResetBeforeFiltering = 0;

% Filter all ones.
sig = ones(100,1);
y1 = [filter(c,sig(1:50)); filter(c,sig(51:end))];
```

```
% Reset the channel and filter all ones.
reset(c);          % Generate an independent channel
y2 = [filter(c,sig(1:50)); filter(c,sig(51:end))];

% Plot the magnitude of the channel output
plot(abs([y1; y2]),'*')
grid on
```

This example generates the following figure.



See Also

`rayleighchan` | `riciananchan` | `filter`

How To

- "Fading Channels"

Purpose

Reset equalizer object

Syntax

```
reset(eqobj)
```

Description

`reset(eqobj)` resets the equalizer object `eqobj`, initializing the `Weights`, `WeightInputs`, and `NumSamplesProcessed` properties and the adaptive algorithm states. If `eqobj` is a CMA equalizer, `reset` does not change the `Weights` property.

See Also

`dfe` | `equalize` | `lineareq`

How To

- "Equalization"

Purpose

Construct Rician fading channel object

Syntax

```
chan = ricianchan(ts,fd,k)
chan = ricianchan(ts,fd,k,tau,pdb)
chan = ricianchan(ts,fd,k,tau,pdb,fdLOS)
chan = ricianchan
```

Description

`chan = ricianchan(ts,fd,k)` constructs a frequency-flat (single path) Rician fading-channel object. `ts` is the sample time of the input signal, in seconds. `fd` is the maximum Doppler shift, in hertz. `k` is the Rician K-factor in linear scale. You can model the effect of the channel `chan` on a signal `x` by using the syntax `y = filter(chan,x)`. See `filter` for more information.

`chan = ricianchan(ts,fd,k,tau,pdb)` constructs a frequency-selective (multiple paths) fading-channel object. If `k` is a scalar, then the first discrete path is a Rician fading process (it contains a line-of-sight component) with a K-factor of `k`, while the remaining discrete paths are independent Rayleigh fading processes (no line-of-sight component). If `k` is a vector of the same size as `tau`, then each discrete path is a Rician fading process with a K-factor given by the corresponding element of the vector `k`. `tau` is a vector of path delays, each specified in seconds. `pdb` is a vector of average path gains, each specified in dB.

`chan = ricianchan(ts,fd,k,tau,pdb,fdLOS)` specifies `fdlos` as the Doppler shift(s) of the line-of-sight component(s) of the discrete path(s), in hertz. `fdlos` must be the same size as `k`. If `k` and `fdlos` are scalars, the line-of-sight component of the first discrete path has a Doppler shift of `fdlos`, while the remaining discrete paths are independent Rayleigh fading processes. If `fdlos` is a vector of the same size as `k`, the line-of-sight component of each discrete path has a Doppler shift given by the corresponding element of the vector `fdlos`. By default, `fdlos` is `0`. The initial phase(s) of the line-of-sight component(s) can be set through the property `DirectPathInitPhase`.

`chan = ricianchan` sets the maximum Doppler shift to `0`, the Rician K-factor to `1`, and the Doppler shift and initial phase of the line-of-sight component to 0. This syntax models a static frequency-flat channel, and, in this trivial case, the sample time of the signal is unimportant.

Properties

The following tables describe the properties of the channel object, chan, that you can set and that MATLAB technical computing software sets automatically. To learn how to view or change the values of a channel object, see "Display Object Properties" or "Change Object Properties".

Writeable Properties

Property	Description
InputSamplePeriod	Sample period of the signal on which the channel acts, measured in seconds.
DopplerSpectrum	Doppler spectrum object(s). The default is a Jakes doppler object.
MaxDopplerShift	Maximum Doppler shift of the channel, in hertz (applies to all paths of a channel).
KFactor	Rician K-factor (scalar or vector). The default value is 1 (line-of-sight component on the first path only).
PathDelays	Vector listing the delays of the discrete paths, in seconds.
AvgPathGaindB	Vector listing the average gain of the discrete paths, in decibels.
DirectPathDopplerShift	Doppler shift(s) of the line-of-sight component(s) in hertz. The default value is 0.

Writeable Properties (Continued)

Property	Description
DirectPathInitPhase	Initial phase(s) of line-of-sight component(s) in radians. The default value is 0.
NormalizePathGains	If this value is 1, the Rayleigh fading process is normalized such that the expected value of the path gains' total power is 1.
StoreHistory	If this value is 1, channel state information needed by the channel visualization tool is stored as the channel filter function processes the signal. The default value is 0.
StorePathGains	If this value is 1, the complex path gain vector is stored as the channel filter function processes the signal. The default value is 0.
ResetBeforeFiltering	If this value is 1, each call to filter resets the state of chan before filtering. If it is 0, the fading process maintains continuity from one call to the next.

Read-Only Properties

Property	Description	When MATLAB Sets or Updates Value
`ChannelType`	Fixed value, `'Rician'`.	When you create object.
`PathGains`	Complex vector listing the current gains of the discrete paths. When you create or reset `chan`, `PathGains` is a random vector influenced by `AvgPathGaindB` and `NormalizePathGains`.	When you create object, reset object, or use it to filter a signal.
`ChannelFilterDelay`	Delay of the channel filter, measured in samples. The ChannelFilterDelay property returns a delay value that is valid only if the first value of the PathGain is the biggest path gain. In other words, main channel energy is in the first path.	When you create object or change ratio of `InputSamplePeriod` to `PathDelays`.
`NumSamplesProcessed`	Number of samples the channel processed since the last reset. When you create or reset `chan`, this property value is `0`.	When you create object, reset object, or use it to filter a signal.

Relationships Among Properties

Changing the length of `PathDelays` also changes the length of `AvgPathGaindB`, the length of `KFactor` if `KFactor` is a vector (no change if it is a scalar), and the length of `DopplerSpectrum` if `DopplerSpectrum` is a vector (no change if it is a single object).

`DirectPathDopplerShift` and `DirectPathInitPhase` both follow changes in `KFactor`.

The `PathDelays` and `AvgPathGaindB` properties of the channel object must always have the same vector length, because this length equals the number of discrete paths of the channel. The `DopplerSpectrum` property must either be a single Doppler object or a vector of Doppler objects with the same length as `PathDelays`.

If you change the length of `PathDelays`, MATLAB truncates or zero-pads the value of `AvgPathGaindB` if necessary to adjust its vector length (MATLAB may also change the values of read-only properties such as `PathGains` and `ChannelFilterDelay`). If `DopplerSpectrum` is a vector of Doppler objects, and you increase or decrease the length of `PathDelays`, MATLAB will add Jakes Doppler objects or remove elements from `DopplerSpectrum`, respectively, to make it the same length as `PathDelays`.

If `StoreHistory` is set to `1` (the default is `0`), the object stores channel state information as the channel filter function processes the signal. You can then visualize this state information through a GUI using the `plot (channel)` method.

Note Setting `StoreHistory` to `1` will result in a slower simulation. If you do not want to visualize channel state information using `plot (channel)`, but want to access the complex path gains, then set `StorePathGains` to `1`, while keeping `StoreHistory` as `0`.

Reset Method

If `MaxDopplerShift` is set to `0` (the default), the channel object, `chan`, models a static channel.

Use the syntax `reset(chan)` to generate a new channel realization.

Algorithm

The methodology used to simulate fading channels is described in "Methodology for Simulating Multipath Fading Channels:", where the properties specific to the Rician channel object are related to the quantities of this section as follows (see the `rayleighchan` reference page for properties common to both Rayleigh and Rician channel objects):

- The `Kfactor` property contains the value of K_r (if it's a scalar) or $\left\{K_{r,k}\right\}$, $1 \le k \le K$ (if it's a vector).
- The `DirectPathDopplerShift` property contains the value of $f_{d,LOS}$ (if it's a scalar) or $\left\{f_{d,LOS,k}\right\}$, $1 \le k \le K$ (if it's a vector).
- The `DirectPathInitPhase` property contains the value of θ_{LOS} (if it's a scalar) or $\left\{\theta_{LOS,k}\right\}$, $1 \le k \le K$ (if it's a vector).

Channel Visualization

The characteristics of a channel can be plotted using the channel visualization tool. You can use the channel visualization tool in Normal mode and Accelerator mode. For more information, see "Channel Visualization".

Examples

The example in "Quasi-Static Channel Modeling" uses this function.

References

[1] Jeruchim, M., Balaban, P., and Shanmugan, K., *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

See Also

`rayleighchan` | `filter` | `plot (channel)` | `reset`

How To

- "Fading Channels"

Purpose

Construct recursive least squares (RLS) adaptive algorithm object

Syntax

```
alg = rls(forgetfactor)
alg = rls(forgetfactor,invcorr0)
```

Description

The `rls` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see "Adaptive Algorithms".

`alg = rls(forgetfactor)` constructs an adaptive algorithm object based on the recursive least squares (RLS) algorithm. The forgetting factor is `forgetfactor`, a real number between 0 and 1. The inverse correlation matrix is initialized to a scalar value.

`alg = rls(forgetfactor,invcorr0)` sets the initialization parameter for the inverse correlation matrix. This scalar value is used to initialize or reset the diagonal elements of the inverse correlation matrix.

Properties

The table below describes the properties of the RLS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see "Access Properties of an Adaptive Algorithm".

Property	Description
`AlgType`	Fixed value, `'RLS'`
`ForgetFactor`	Forgetting factor
`InvCorrInit`	Scalar value used to initialize or reset the diagonal elements of the inverse correlation matrix

Also, when you use this adaptive algorithm object to create an equalizer object (via the `lineareq` function or `dfe` function), the equalizer object has an `InvCorrMatrix` property that represents the inverse correlation matrix for the RLS algorithm. The initial value of `InvCorrMatrix` is `InvCorrInit*eye(N)`, where `N` is the total number of equalizer weights.

Examples

For examples that use this function, see "Defining an Equalizer Object" and "Example: Adaptive Equalization Within a Loop".

Algorithms

Referring to the schematics presented in "Equalizer Structure", define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of inputs, u, and the current inverse correlation matrix, P, this adaptive algorithm first computes the Kalman gain vector, K

$$K = \frac{Pu}{(\texttt{ForgetFactor}) + u^H Pu}$$

where H denotes the Hermitian transpose.

Then the new inverse correlation matrix is given by

$$(\texttt{ForgetFactor})^{-1}(\mathrm{P} - \mathrm{K}\mathrm{u}^{\mathrm{H}}\mathrm{P})$$

and the new set of weights is given by

$$w + K^{*}\mathrm{e}$$

where the * operator denotes the complex conjugate.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, John Wiley & Sons, 1998.

[2] Haykin, S., *Adaptive Filter Theory*, Third Ed., Upper Saddle River, NJ, Prentice-Hall, 1996.

[3] Kurzweil, J., *An Introduction to Digital Communications*, New York, John Wiley & Sons, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

See Also

`lms` | `signlms` | `normlms` | `varlms` | `lineareq` | `dfe` | `equalize`

How To

- "Equalization"

Purpose

Reed-Solomon decoder

Syntax

```
decoded = rsdec(code,n,k)
decoded = rsdec(code,n,k,genpoly)
decoded = rsdec(...,paritypos)
[decoded,cnumerr] = rsdec(...)
[decoded,cnumerr,ccode] = rsdec(...)
```

Description

`decoded = rsdec(code,n,k)` attempts to decode the received signal in `code` using an [n,k] Reed-Solomon decoding process with the narrow-sense generator polynomial. `code` is a Galois array of symbols having m bits each. Each `n`-element row of `code` represents a corrupted systematic codeword, where the parity symbols are at the end and the leftmost symbol is the most significant symbol. `n` is at most 2^m-1. If `n` is not exactly 2^m-1, `rsdec` assumes that `code` is a corrupted version of a shortened code.

In the Galois array `decoded`, each row represents the attempt at decoding the corresponding row in `code`. A *decoding failure* occurs if `rsdec` detects more than `(n-k)/2` errors in a row of `code`. In this case, `rsdec` forms the corresponding row of `decoded` by merely removing `n-k` symbols from the end of the row of `code`.

`decoded = rsdec(code,n,k,genpoly)` is the same as the syntax above, except that a nonempty value of `genpoly` specifies the generator polynomial for the code. In this case, `genpoly` is a Galois row vector that lists the coefficients, in order of descending powers, of the generator polynomial. The generator polynomial must have degree `n-k`. To use the default narrow-sense generator polynomial, set `genpoly` to `[]`.

`decoded = rsdec(...,`*`paritypos`*`)` specifies whether the parity symbols in `code` were appended or prepended to the message in the coding operation. The string *`paritypos`* can be either `'end'` or `'beginning'`. The default is `'end'`. If *`paritypos`* is `'beginning'`, a decoding failure causes `rsdec` to remove `n-k` symbols from the beginning rather than the end of the row.

`[decoded,cnumerr] = rsdec(...)` returns a column vector `cnumerr`, each element of which is the number of corrected errors in the corresponding row of `code`. A value of `-1` in `cnumerr` indicates a decoding failure in that row in `code`.

`[decoded,cnumerr,ccode] = rsdec(...)` returns `ccode`, the corrected version of `code`. The Galois array `ccode` has the same format as `code`. If a decoding failure occurs in a certain row of `code`, the corresponding row in `ccode` contains that row unchanged.

Examples

The example below encodes three message words using a (7,3) Reed-Solomon encoder. It then corrupts the code by introducing one error in the first codeword, two errors in the second codeword, and three errors in the third codeword. Then `rsdec` tries to decode the corrupted code.

```
m = 3; % Number of bits per symbol
n = 2^m-1; k = 3; % Word lengths for code
msg = gf([2 7 3; 4 0 6; 5 1 1],m); % Three rows of m-bit symbols
code = rsenc(msg,n,k);
errors = gf([2 0 0 0 0 0 0; 3 4 0 0 0 0 0; 5 6 7 0 0 0 0],m);
noisycode = code + errors;
[dec,cnumerr] = rsdec(noisycode,n,k)
```

The output is below.

```
dec = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     2     7     3
     4     0     6
     0     7     6


cnumerr =

     1
     2
    -1
```

The output shows that `rsdec` successfully corrects the errors in the first two codewords and recovers the first two original message words. However, a (7,3) Reed-Solomon code can correct at most two errors in each word, so `rsdec` cannot recover the third message word. The elements of the vector `cnumerr` indicate the number of corrected errors in the first two words and also indicate the decoding failure in the third word.

For additional examples, see "Create and Decode Reed-Solomon Codes".

Limitations

`n` and `k` must differ by an even integer. `n` must be between 3 and 65535.

Algorithms

`rsdec` uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the works listed in "References" on page 1-482 below.

References

[1] Wicker, S. B., *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, NJ, Prentice Hall, 1995.

[2] Berlekamp, E. R., *Algebraic Coding Theory*, New York, McGraw-Hill, 1968.

See Also

`rsenc` | `gf` | `rsgenpoly`

How To

- "Block Codes"

Purpose

Decode ASCII file encoded using Reed-Solomon code

Syntax

```
rsdecof(file_in,file_out); rsdecof(file_in,file_out,err_cor);
```

Description

Warning

This function is obsolete and may be removed in the future. We strongly recommend that you use the `comm.RSDecoder` System object instead.

This function is the inverse process of the function `rsencof` in that it decodes a file that `rsencof` encoded.

`rsdecof(file_in,file_out)` decodes the ASCII file `file_in` that was previously created by the function `rsencof` using an error-correction capability of 5. The decoded message is written to `file_out`. Both `file_in` and `file_out` are string variables.

Note If the number of characters in `file_in` is not an integer multiple of 127, the function appends `char(4)` symbols to the data it must decode. If you encode and then decode a file using `rsencof` and `rsdecof`, respectively, the decoded file might have `char(4)` symbols at the end that the original file does not have.

`rsdecof(file_in,file_out,err_cor)` is the same as the first syntax, except that `err_cor` specifies the error-correction capability for each block of 127 codeword characters. The message length is 127 - 2 `*err_cor`. The value in `err_cor` must match the value used in `rsencof` when `file_in` was created.

Examples

An example is on the reference page for `rsencof`.

See Also

`rsencof`

How To

- "Block Codes"

Purpose

Reed-Solomon encoder

Syntax

```
code = rsenc(msg,n,k)
code = rsenc(msg,n,k,genpoly)
code = rsenc(...,paritypos)
```

Description

`code = rsenc(msg,n,k)` encodes the message in `msg` using an [`n`,`k`] Reed-Solomon code with the narrow-sense generator polynomial. `msg` is a Galois array of symbols having m bits each. Each `k`-element row of `msg` represents a message word, where the leftmost symbol is the most significant symbol. `n` is at most 2^m-1. If `n` is not exactly 2^m-1, `rsenc` uses a shortened Reed-Solomon code. Parity symbols are at the end of each word in the output Galois array `code`.

`code = rsenc(msg,n,k,genpoly)` is the same as the syntax above, except that a nonempty value of `genpoly` specifies the generator polynomial for the code. In this case, `genpoly` is a Galois row vector that lists the coefficients, in order of descending powers, of the generator polynomial. The generator polynomial must have degree `n-k`. To use the default narrow-sense generator polynomial, set `genpoly` to `[]`.

`code = rsenc(...,`*`paritypos`*`)` specifies whether `rsenc` appends or prepends the parity symbols to the input message to form `code`. The string *`paritypos`* can be either `'end'` or `'beginning'`. The default is `'end'`.

Examples

The example below encodes two message words using a (7,3) Reed-Solomon encoder.

```
m = 3; % Number of bits per symbol
n = 2^m-1; k = 3; % Word lengths for code
msg = gf([2 7 3; 4 0 6],m); % Two rows of m-bit symbols
code = rsenc(msg,n,k)
```

The output is below.

```
code = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
```

```
Array elements =

     2     7     3     3     6     7     6
     4     0     6     4     2     2     0
```

For additional examples, see “Represent Words for Reed-Solomon Codes” and “Create and Decode Reed-Solomon Codes”.

Limitations

`n` and `k` must differ by an even integer. `n` must be between 3 and 65535.

See Also

`rsdec` | `gf` | `rsgenpoly`

How To

- “Block Codes”

Purpose

Encode ASCII file using Reed-Solomon code

Syntax

`rsencof(file_in,file_out); rsencof(file_in,file_out,err_cor);`

Description

Warning

This function is obsolete and may be removed in the future. We strongly recommend that you use the `comm.RSEncoder` System object instead.

`rsencof(file_in,file_out)` encodes the ASCII file `file_in` using (127, 117) Reed-Solomon code. The error-correction capability of this code is 5 for each block of 127 codeword characters. This function writes the encoded text to the file `file_out`. Both `file_in` and `file_out` are string variables.

`rsencof(file_in,file_out,err_cor)` is the same as the first syntax, except that `err_cor` specifies the error-correction capability for each block of 127 codeword characters. The message length is 127 - 2 * `err_cor`.

Note If the number of characters in `file_in` is not an integer multiple of 127 - 2 * `err_cor`, the function appends `char(4)` symbols to `file_out`.

Examples

The file *`matlabroot`*`/toolbox/comm/comm/oct2dec.m` contains text help for the `oct2dec` function in this toolbox. The commands below encode the file using `rsencof` and then decode it using `rsdecof`.

```
file_in = [matlabroot '/toolbox/comm/comm/oct2dec.m'];
file_out = 'encodedfile'; % Or use another filename
rsencof(file_in,file_out) % Encode the file.

file_in = file_out;
file_out = 'decodedfile'; % Or use another filename
rsdecof(file_in,file_out) % Decode the file.
```

To see the original file and the decoded file in the MATLAB workspace, use the commands below (or similar ones if you modified the filenames above).

```
type oct2dec.m
type decodedfile
```

See Also

`rsdecof`

How To

- "Block Codes"

rsgenpoly

Purpose Generator polynomial of Reed-Solomon code

Syntax

```
genpoly = rsgenpoly(n,k)
genpoly = rsgenpoly(n,k,prim_poly)
genpoly = rsgenpoly(n,k,prim_poly,b)
[genpoly,t] = rsgenpoly(...)
```

Description

`genpoly = rsgenpoly(n,k)` returns the narrow-sense generator polynomial of a Reed-Solomon code with codeword length `n` and message length `k`. The codeword length `n` must have the form 2^m-1 for some integer m between 3 and 16.

, and `n-k` must be an even integer. The output `genpoly` is a Galois row vector that represents the coefficients of the generator polynomial in order of descending powers. The narrow-sense generator polynomial is (X - $Alpha^1$)(X - $Alpha^2$)...(X - $Alpha^{2t}$) where:

- Alpha represents a root of the default primitive polynomial for the field GF(`n+1`),
- and t represents the code's error-correction capability, `(n-k)/2`.

`genpoly = rsgenpoly(n,k,prim_poly)` is the same as the syntax above, except that `prim_poly` specifies the primitive polynomial for GF(`n+1`) that has Alpha as a root. `prim_poly` is an integer whose binary representation indicates the coefficients of the primitive polynomial. To use the default primitive polynomial GF(`n+1`), set `prim_poly` to `[]`.

`genpoly = rsgenpoly(n,k,prim_poly,b)` returns the generator polynomial (X - $Alpha^b$)(X - $Alpha^{b+1}$)...(X - $Alpha^{b+2t-1}$), where:

- `b` is an integer,
- Alpha is a root of `prim_poly`,
- and t is the code's error-correction capability, `(n-k)/2`.

`[genpoly,t] = rsgenpoly(...)` returns `t`, the error-correction capability of the code.

Examples

The examples below create Galois row vectors that represent generator polynomials for a [7,3] Reed-Solomon code. The vectors g and g2 both represent the narrow-sense generator polynomial, but with respect to different primitive elements A. More specifically, g2 is defined such that A is a root of the primitive polynomial $D^3 + D^2 + 1$ for GF(8), not of the default primitive polynomial $D^3 + D + 1$. The vector g3 represents the generator polynomial $(X - A^3)(X - A^4)(X - A^5)(X - A^6)$, where A is a root of $D^3 + D^2 + 1$ in GF(8).

```
g = rsgenpoly(7,3)
g2 = rsgenpoly(7,3,13) % Use nondefault primitive polynomial.
g3 = rsgenpoly(7,3,13,3) % Use b = 3.
```

The output is below.

```
g = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     1     3     1     2     3


g2 = GF(2^3) array. Primitive polynomial = D^3+D^2+1 (13 decimal)

Array elements =

     1     4     5     1     5


g3 = GF(2^3) array. Primitive polynomial = D^3+D^2+1 (13 decimal)

Array elements =

     1     7     1     6     7
```

As another example, the command below shows that the default narrow-sense generator polynomial for a [15,11] Reed-Solomon code is

$X^4 + (A^3 + A^2 + 1)X^3 + (A^3 + A^2)X^2 + A^3X + (A^2 + A + 1)$, where A is a root of the default primitive polynomial for GF(16).

```
gp = rsgenpoly(15,11)

gp = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     1    13    12     8     7
```

For additional examples, see “Parameters for Reed-Solomon Codes”.

Limitations

`n` and `k` must differ by an even integer. The maximum allowable value of `n` is 65535.

See Also

`gf` | `rsenc` | `rsdec`

How To

- “Block Codes”

rsgenpolycoeffs

Purpose

Generator polynomial coefficients of Reed-Solomon code

Syntax

```
x = rsgenpolycoeffs(...)
[x,t] = rsgenpolycoeffs(...)
```

Description

`x = rsgenpolycoeffs(...)` returns the coefficients for the generator polynomial of the Reed-Solomon code. The output is identical to genpoly = rsgenpoly(...); x = genpoly.x.

`[x,t] = rsgenpolycoeffs(...)` returns *t*, the error-correction capability of the code.

See Also

`rsgenpoly` | `gf` | `rsenc` | `rsdec`

Purpose

Generate scatter plot

Syntax

```
scatterplot(x)
scatterplot(x,n)
scatterplot(x,n,offset)
scatterplot(x,n,offset,plotstring)
scatterplot(x,n,offset,plotstring,h)
h = scatterplot(...)
```

Description

`scatterplot(x)` produces a scatter plot for the signal `x`. The interpretation of `x` depends on its shape and complexity:

- If `x` is a real two-column matrix, `scatterplot` interprets the first column as in-phase components and the second column as quadrature components.
- If `x` is a complex vector, `scatterplot` interprets the real part as in-phase components and the imaginary part as quadrature components.
- If `x` is a real vector, `scatterplot` interprets it as a real signal.

`scatterplot(x,n)` is the same as the first syntax, except that the function plots every `n`th value of the signal, starting from the first value. That is, the function decimates `x` by a factor of `n` before plotting.

`scatterplot(x,n,offset)` is the same as the first syntax, except that the function plots every `n`th value of the signal, starting from the (`offset+1`)st value in `x`.

`scatterplot(x,n,offset,plotstring)` is the same as the syntax above, except that `plotstring` determines the plotting symbol, line type, and color for the plot. `plotstring` is a string whose format and meaning are the same as in the `plot` function.

`scatterplot(x,n,offset,plotstring,h)` is the same as the syntax above, except that the scatter plot is in the figure whose handle is `h`, rather than a new figure. `h` must be a handle to a figure that `scatterplot` previously generated. To plot multiple signals in the same figure, use `hold on`.

`h = scatterplot(...)` is the same as the earlier syntaxes, except that `h` is the handle to the figure that contains the scatter plot.

Examples

See "View Signals Using Scatter Plots" or the example on the reference page for `qamdemod`. Both examples illustrate how to plot multiple signals in a single scatter plot.

For an online demonstration, type `showdemo scattereyedemo`.

See Also

`eyediagram` | `plot` | `scatter`

How To

- scattereyedemo
- "Scatter Plots"

Purpose Calculate bit error rate (BER) using semianalytic technique

Syntax

```
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp)
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,num,den)
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,EbNo)
ber =
semianalytic(txsig,rxsig,modtype,M,Nsamp,num,den,EbNo)
[ber,avgampl,avgpower] = semianalytic(...)
```

Alternatives As an alternative to the `semianalytic` function, invoke the BERTool GUI (`bertool`) and use the **Semianalytic** tab.

Description `ber = semianalytic(txsig,rxsig,`*`modtype`*`,`*`M`*`,Nsamp)` returns the bit error rate (BER) of a system that transmits the complex baseband vector signal `txsig` and receives the noiseless complex baseband vector signal `rxsig`. Each of these signals has `Nsamp` samples per symbol. `Nsamp` is also the sampling rate of `txsig` and `rxsig`, in Hz. The function assumes that `rxsig` is the input to the receiver filter, and the function filters `rxsig` with an ideal integrator. `modtype` is the modulation type of the signal and *`M`* is the alphabet size. The table below lists the valid values for *`modtype`* and *`M`*.

Modulation Scheme	Value of *modtype*	Valid Values of *M*
Differential phase shift keying (DPSK)	`'dpsk'`	2, 4
Minimum shift keying (MSK) with differential encoding	`'msk/diff'`	2
Minimum shift keying (MSK) with nondifferential encoding	`'msk/nondiff'`	2

Modulation Scheme	Value of *modtype*	Valid Values of *M*
Phase shift keying (PSK) with differential encoding, where the phase offset of the constellation is 0	`'psk/diff'`	2, 4
Phase shift keying (PSK) with nondifferential encoding, where the phase offset of the constellation is 0	`'psk/nondiff'`	2, 4, 8, 16, 32, or 64
Offset quaternary phase shift keying (OQPSK)	`'oqpsk'`	4
Quadrature amplitude modulation (QAM)	`'qam'`	4, 8, 16, 32, 64, 128, 256, 512, 1024

`'msk/diff'` is equivalent to conventional MSK (setting the `'Precoding'` property of the MSK object to `'off'`), while `'msk/nondiff'` is equivalent to precoded MSK (setting the `'Precoding'` property of the MSK object to `'on'`).

Note The output `ber` is an *upper bound* on the BER in these cases:

- DQPSK (*modtype* = `'dpsk'`, `M` = 4)
- Cross QAM (*modtype* = `'qam'`, `M` not a perfect square). In this case, note that the upper bound used here is slightly tighter than the upper bound used for cross QAM in the `berawgn` function.

When the function computes the BER, it assumes that symbols are Gray-coded. The function calculates the BER for values of E_b/N_0 in the range of [0:20] dB and returns a vector of length 21 whose elements correspond to the different E_b/N_0 levels.

Note You must use a sufficiently long vector `txsig`, or else the calculated BER will be inaccurate. If the system's impulse response is L symbols long, the length of `txsig` should be at least M^L. A common approach is to start with an augmented binary pseudonoise (PN) sequence of total length $(\log_2 M)M^L$. An *augmented* PN sequence is a PN sequence with an extra zero appended, which makes the distribution of ones and zeros equal.

`ber = semianalytic(txsig,rxsig,`*`modtype`*`,`*`M`*`,Nsamp,num,den)` is the same as the previous syntax, except that the function filters `rxsig` with a receiver filter instead of an ideal integrator. The transfer function of the receiver filter is given in descending powers of z by the vectors `num` and `den`.

`ber = semianalytic(txsig,rxsig,`*`modtype`*`,`*`M`*`,Nsamp,EbNo)` is the same as the first syntax, except that `EbNo` represents E_b/N_0, the ratio of bit energy to noise power spectral density, in dB. If `EbNo` is a vector, then the output `ber` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels.

`ber =`
`semianalytic(txsig,rxsig,`*`modtype`*`,`*`M`*`,Nsamp,num,den,EbNo)` combines the functionality of the previous two syntaxes.

`[ber,avgampl,avgpower] = semianalytic(...)` returns the mean complex signal amplitude and the mean power of `rxsig` after filtering it by the receiver filter and sampling it at the symbol rate.

Examples

A typical procedure for implementing the semianalytic technique is in “Procedure for the Semianalytic Technique”. Sample code is in “Example: Using the Semianalytic Technique”.

Limitations

The function makes several important assumptions about the communication system. See “When to Use the Semianalytic Technique” to find out whether your communication system is suitable for the semianalytic technique and the `semianalytic` function.

References

[1] Jeruchim, M. C., P. Balaban, and K. S. Shanmugan, *Simulation of Communication Systems*, New York, Plenum Press, 1992.

[2] Pasupathy, S., “Minimum Shift Keying: A Spectrally Efficient Modulation,” *IEEE Communications Magazine*, July, 1979, pp. 14–22.

See Also

`noisebw` | `qfunc`

How To

- “Performance Results via the Semianalytic Technique”

shift2mask

Purpose

Convert shift to mask vector for shift register configuration

Syntax

```
mask = shift2mask(prpoly,shift)
```

Description

`mask = shift2mask(prpoly,shift)` returns the mask that is equivalent to the shift (or offset) specified by `shift`, for a linear feedback shift register whose connections are specified by the primitive polynomial `prpoly`. The `prpoly` input can have one of these formats:

- A binary vector that lists the coefficients of the primitive polynomial in order of descending powers
- An integer scalar whose binary representation gives the coefficients of the primitive polynomial, where the least significant bit is the constant term

The `shift` input is an integer scalar.

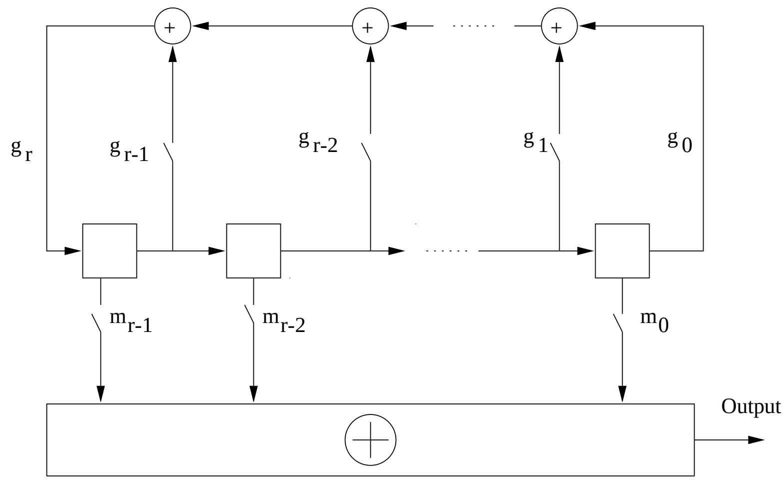
Note To save time, `shift2mask` does not check that `prpoly` is primitive. If it is not primitive, the output is not meaningful. To find primitive polynomials, use `primpoly` or see [2].

Definition of Equivalent Mask

The equivalent mask for the shift s is the remainder after dividing the polynomial x^s by the primitive polynomial. The vector `mask` represents the remainder polynomial by listing the coefficients in order of descending powers.

Shifts, Masks, and Pseudonoise Sequence Generators

Linear feedback shift registers are part of an implementation of a pseudonoise sequence generator. Below is a schematic diagram of a pseudonoise sequence generator. All adders perform addition modulo 2.



The primitive polynomial determines the state of each switch labeled g_k, and the mask determines the state of each switch labeled m_k. The lower half of the diagram shows the implementation of the shift, which delays the starting point of the output sequence. If the shift is zero, the m_0 switch is closed while all other m_k switches are open. The table below indicates how the shift affects the shift register's output.

	T = 0	**T = 1**	**T = 2**	**...**	**T = s**	**T = s+1**
Shift = 0	x_0	x_1	x_2	...	x_s	x_{s+1}
Shift = s > 0	x_s	x_{s+1}	x_{s+2}	...	x_{2s}	x_{2s+1}

If you have Communications System Toolbox software and want to generate a pseudonoise sequence in a Simulink model, see the PN Sequence Generator block reference page.

Examples

The command below converts a shift of 5 into the equivalent mask $x^3 + x + 1$, for the linear feedback shift register whose connections are specified by the primitive polynomial $x^4 + x^3 + 1$.

```
mk = shift2mask([1 1 0 0 1],5)

mk =

     1     0     1     1
```

References

[1] Lee, J. S., and L. E. Miller, *CDMA Systems Engineering Handbook*, Boston, Artech House, 1998.

[2] Simon, Marvin K., Jim K. Omura, et al., *Spread Spectrum Communications Handbook*, New York, McGraw-Hill, 1994.

See Also

`mask2shift` | `deconv` | `isprimitive` | `primpoly`

Purpose

Construct signed least mean square (LMS) adaptive algorithm object

Syntax

```
alg = signlms(stepsize)
alg = signlms(stepsize,algtype)
```

Description

The `signlms` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see "Adaptive Algorithms".

`alg = signlms(stepsize)` constructs an adaptive algorithm object based on the signed least mean square (LMS) algorithm with a step size of `stepsize`.

`alg = signlms(stepsize,`*`algtype`*`)` constructs an adaptive algorithm object of type *`algtype`* from the family of signed LMS algorithms. The table below lists the possible values of *`algtype`*.

Value of *`algtype`*	Type of Signed LMS Algorithm
`'Sign LMS'`	Sign LMS (default)
`'Signed Regressor LMS'`	Signed regressor LMS
`'Sign Sign LMS'`	Sign-sign LMS

Properties

The table below describes the properties of the signed LMS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see "Access Properties of an Adaptive Algorithm".

Property	Description
`AlgType`	Type of signed LMS algorithm, corresponding to the *`algtype`* input argument. You cannot change the value of this property after creating the object.
`StepSize`	LMS step size parameter, a nonnegative real number
`LeakageFactor`	LMS leakage factor, a real number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, while a value of 0 corresponds to a memoryless update algorithm.

Algorithms

Referring to the schematics presented in "Equalizer Structure", define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of weights, w, this adaptive algorithm creates the new set of weights given by

- `(LeakageFactor) w + (StepSize) u*sgn(Re(e))`, for sign LMS
- `(LeakageFactor) w + (StepSize) sgn(Re(u)) Re(e)`, for signed regressor LMS
- `(LeakageFactor) w + (StepSize) sgn(Re(u)) sgn(Re(e))`, for sign-sign LMS

where the `*` operator denotes the complex conjugate and `sgn` denotes the signum function (`sign` in MATLAB technical computing software).

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, John Wiley & Sons, 1998.

[2] Kurzweil, J., *An Introduction to Digital Communications*, New York, John Wiley & Sons, 2000.

See Also

`lms` | `normlms` | `varlms` | `rls` | `cma` | `lineareq` | `dfe` | `equalize`

How To

- "Equalization"

Purpose

Single sideband amplitude demodulation

Syntax

```
z = ssbdemod(y,Fc,Fs)
z = ssbdemod(y,Fc,Fs,ini_phase)
z = ssbdemod(y,Fc,Fs,ini_phase,num,den)
```

Description

For All Syntaxes

`z = ssbdemod(y,Fc,Fs)` demodulates the single sideband amplitude modulated signal `y` from the carrier signal having frequency `Fc` (Hz). The carrier signal and `y` have sampling rate `Fs` (Hz). The modulated signal has zero initial phase, and can be an upper- or lower-sideband signal. The demodulation process uses the lowpass filter specified by `[num,den] = butter(5,Fc*2/Fs)`.

Note The `Fc` and `Fs` arguments must satisfy `Fs` > 2(`Fc` + `BW`), where `BW` is the bandwidth of the original signal that was modulated.

`z = ssbdemod(y,Fc,Fs,ini_phase)` specifies the initial phase of the modulated signal in radians.

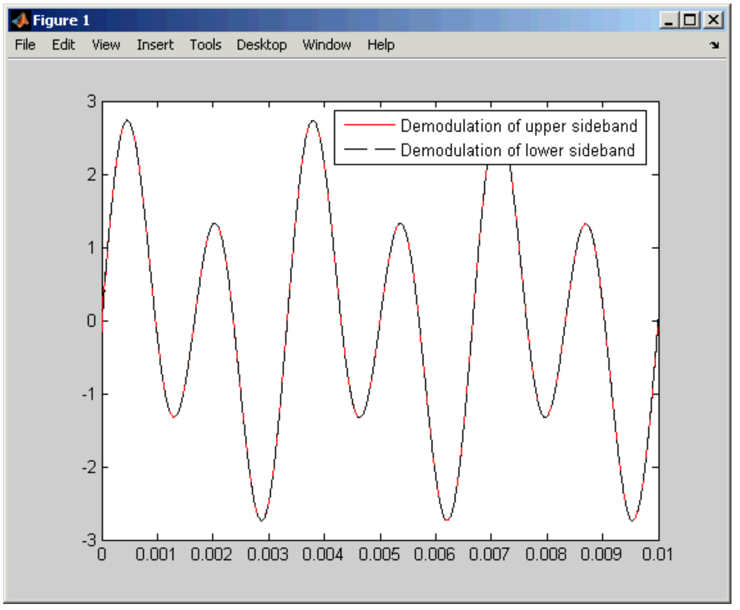
`z = ssbdemod(y,Fc,Fs,ini_phase,num,den)` specifies the numerator and denominator of the lowpass filter used in the demodulation.

Examples

The code below shows that `ssbdemod` can demodulate an upper-sideband or lower-sideband signal.

```
Fc = 12000; Fs = 270000;
t = [0:1/Fs:0.01]';
s = sin(2*pi*300*t)+2*sin(2*pi*600*t);
y1 = ssbmod(s,Fc,Fs,0); % Lower-sideband modulated signal
y2 = ssbmod(s,Fc,Fs,0,'upper'); % Upper-sideband modulated signal
s1 = ssbdemod(y1,Fc,Fs); % Demodulate lower sideband
s2 = ssbdemod(y2,Fc,Fs); % Demodulate upper sideband
% Plot results to show that the curves overlap.
figure; plot(t,s1,'r-',t,s2,'k--');
```

```
legend('Demodulation of upper sideband','Demodulation of lower sideband')
```



See Also

`ssbmod` | `amdemod`

How To

- "Digital Modulation"

Purpose

Single sideband amplitude modulation

Syntax

```
y = ssbmod(x,Fc,Fs)
y = ssbmod(x,Fc,Fs,ini_phase)
y = ssbmod(x,fc,fs,ini_phase,'upper')
```

Description

`y = ssbmod(x,Fc,Fs)` uses the message signal `x` to modulate a carrier signal with frequency `Fc` (Hz) using single sideband amplitude modulation in which the lower sideband is the desired sideband. The carrier signal and `x` have sample frequency `Fs` (Hz). The modulated signal has zero initial phase.

`y = ssbmod(x,Fc,Fs,ini_phase)` specifies the initial phase of the modulated signal in radians.

`y = ssbmod(x,fc,fs,ini_phase,'upper')` uses the upper sideband as the desired sideband.

Examples

An example using `ssbmod` is on the reference page for `ammod`.

See Also

`ssbdemod` | `ammod`

How To

- “Digital Modulation”

Purpose

Construct channel object from set of standardized channel models

Syntax

```
chan = stdchan(ts,fd,chantype)
[chan, chanprofile] = stdchan(...)
chan = stdchan(ts,fd,chantype, trms)
```

Description

`chan = stdchan(ts,fd,chantype)` constructs a fading channel object `chan` according to the specified `chantype`. The input string `chantype` is chosen from the set of standardized channel profiles listed below. `ts` is the sample time of the input signal, in seconds. `fd` is the maximum Doppler shift, in Hertz.

`[chan, chanprofile] = stdchan(...)` also returns a structure `chanprofile` containing the parameters of the channel profile specified by `chantype`.

`chan = stdchan(ts,fd,chantype, trms)` is used to create a channel object, `chan`, when `chantype` is any one of `'802.11a'`, `'802.11b'` or `'802.11g'`. When using `'802.11a'`, `'802.11b'` or `'802.11g'` channels, you must specify TRMS, which is the RMS delay spread of the channel model. As per 802.11 specifications, TS should not be larger than TRMS/2.

Channel Models

COST 207 channel models (The Rician K factors for the cases `cost207RAx4` and `cost207RAx6` are chosen as in 3GPP TS 45.005 V7.9.0 (2007-2)):

Channel model	Profile
`cost207RAx4`	Rural Area (RAx), 4 taps
`cost207RAx6`	Rural Area (RAx), 6 taps
`cost207TUx6`	Typical Urban (TUx), 6 taps
`cost207TUx6alt`	Typical Urban (TUx), 6 taps, alternative
`cost207TUx12`	Typical Urban (TUx), 12 taps

Channel model	Profile
cost207TUx12alt	Typical Urban (TUx), 12 taps, alternative
cost207BUx6	Bad Urban (BUx), 6 taps
cost207BUx6alt	Bad Urban (BUx), 6 taps, alternative
cost207BUx12	Bad Urban (BUx), 12 taps
cost207BUx12alt	Bad Urban (BUx), 12 taps, alternative
cost207HTx6	Hilly Terrain (HTx), 6 taps
cost207HTx6alt	Hilly Terrain (HTx), 6 taps, alternative
cost207HTx12	Hilly Terrain (HTx), 12 taps
cost207HTx12alt	Hilly Terrain (HTx), 12 taps, alternative

GSM/EDGE channel models (3GPP TS 45.005 V7.9.0 (2007-2), 3GPP TS 05.05 V8.20.0 (2005-11)):

Channel model	Profile
gsmRAx6c1	Typical case for rural area (RAx), 6 taps, case 1
gsmRAx4c2	Typical case for rural area (RAx), 4 taps, case 2
gsmHTx12c1	Typical case for hilly terrain (HTx), 12 taps, case 1
gsmHTx12c2	Typical case for hilly terrain (HTx), 12 taps, case 2

Channel model	Profile
gsmHTx6c1	Typical case for hilly terrain (HTx), 6 taps, case 1
gsmHTx6c2	Typical case for hilly terrain (HTx), 6 taps, case 2
gsmTUx12c1	Typical case for urban area (TUx), 12 taps, case 1
gsmTUx12c1	Typical case for urban area (TUx), 12 taps, case 2
gsmTUx6c1	Typical case for urban area (TUx), 6 taps, case 1
gsmTUx6c2	Typical case for urban area (TUx), 6 taps, case 2
gsmEQx6	Profile for equalization test (EQx), 6 taps
gsmTIx2	Typical case for very small cells (TIx), 2 taps

3GPP channel models for deployment evaluation (3GPP TR 25.943 V6.0.0 (2004-12)):

Channel model	Profile
3gppTUx	Typical Urban channel model (TUx)
3gppRAx	Rural Area channel model (RAx)
3gppHTx	Hilly Terrain channel model (HTx)

ITU-R 3G channel models (ITU-R M.1225 (1997-2)):

Channel model	Profile
`itur3GIAx`	Indoor office, channel A
`itur3GIBx`	Indoor office, channel B
`itur3GPAx`	Outdoor to indoor and pedestrian, channel A
`itur3GPBx`	Outdoor to indoor and pedestrian, channel B
`itur3GVAx`	Vehicular - high antenna, channel A
`itur3GVBx`	Vehicular - high antenna, channel B
`itur3GSAxLOS`	Satellite, channel A, LOS
`itur3GSAxNLOS`	Satellite, channel A, NLOS
`itur3GSBxLOS`	Satellite, channel B, LOS
`itur3GSBxNLOS`	Satellite, channel B, NLOS
`itur3GSCxLOS`	Satellite, channel C, LOS
`itur3GSCxNLOS`	Satellite, channel C, NLOS

ITU-R HF channel models (ITU-R F.1487 (2000)) (FD must be 1 to obtain the correct frequency spreads for these models.):

Channel model	Profile
`iturHFLQ`	Low latitudes, Quiet conditions
`iturHFLM`	Low latitudes, Moderate conditions
`iturHFLD`	Low latitudes, Disturbed conditions

Channel model	Profile
`iturHFMQ`	Medium latitudes, Quiet conditions
`iturHFMM`	Medium latitudes, Moderate conditions
`iturHFMD`	Medium latitudes, Disturbed conditions
`iturHFMDV`	Medium latitudes, Disturbed conditions near vertical incidence
`iturHFHQ`	High latitudes, Quiet conditions
`iturHFHM`	High latitudes, Moderate conditions
`iturHFHD`	High latitudes, Disturbed conditions

JTC channel models:

Channel model	Profile
`jtcInResA`	Indoor residential A
`jtcInResB`	Indoor residential B
`jtcInResC`	Indoor residential C
`jtcInOffA`	Indoor office A
`jtcInOffB`	Indoor office B
`jtcInOffC`	Indoor office C
`jtcInComA`	Indoor commercial A
`jtcInComB`	Indoor commercial B
`jtcInComC`	Indoor commercial C

Channel model	Profile
jtcOutUrbHRLAA	Outdoor urban high-rise areas - Low antenna A
jtcOutUrbHRLAB	Outdoor urban high-rise areas - Low antenna B
jtcOutUrbHRLAC	Outdoor urban high-rise areas - Low antenna C
jtcOutUrbLRLAA	Outdoor urban low-rise areas - Low antenna A
jtcOutUrbLRLAB	Outdoor urban low-rise areas - Low antenna B
jtcOutUrbLRLAC	Outdoor urban low-rise areas - Low antenna C
jtcOutResLAA	Outdoor residential areas - Low antenna A
jtcOutResLAB	Outdoor residential areas - Low antenna B
jtcOutResLAC	Outdoor residential areas - Low antenna C
jtcOutUrbHRHAA	Outdoor urban high-rise areas - High antenna A
jtcOutUrbHRHAB	Outdoor urban high-rise areas - High antenna B
jtcOutUrbHRHAC	Outdoor urban high-rise areas - High antenna C
jtcOutUrbLRHAA	Outdoor urban low-rise areas - High antenna A
jtcOutUrbLRHAB	Outdoor urban low-rise areas - High antenna B

Channel model	Profile
jtcOutUrbLRHAC	Outdoor urban low-rise areas - High antenna C
jtcOutResHAA	Outdoor residential areas - High antenna A
jtcOutResHAB	Outdoor residential areas - High antenna B
jtcOutResHAC	Outdoor residential areas - High antenna C

HIPERLAN/2 channel models:

Channel model	Profile
hiperlan2A	Model A
hiperlan2B	Model B
hiperlan2C	Model C
hiperlan2D	Model D
hiperlan2E	Model E

802.11a/b/g channel models:

802.11a/b/g channel models share a common multipath delay profile

Note TS should not be larger than TRMS/2, as per 802.11 specifications.

Channel model
802.11a
802.11b
802.11g

Examples

```
ts = 0.1e-4; fd = 200;
chan = stdchan(ts, fd, 'cost207TUx6');
chan.NormalizePathGains = 1;
chan.StoreHistory = 1;
y = filter(chan, ones(1,5e4));
plot(chan);
```

See Also

doppler | rayleighchan | ricianchan

Purpose

Compute number of symbol errors and symbol error rate

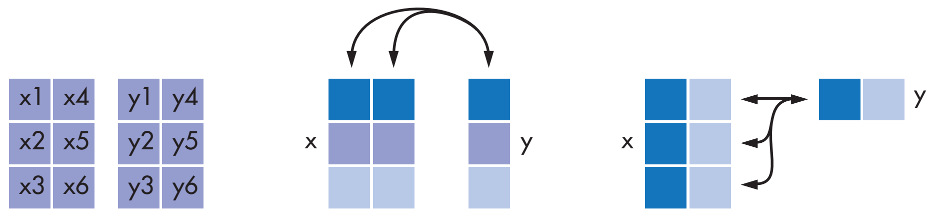
Syntax

```
[number,ratio] = symerr(x,y)
[number,ratio] = symerr(x,y,flg)
[number,ratio,loc] = symerr(...)
```

Description

For All Syntaxes

The `symerr` function compares binary representations of elements in `x` with those in `y`. The schematics below illustrate how the shapes of `x` and `y` determine which elements `symerr` compares.



(a) Compares x1 with y1, x2 with y2, and so on.
(b) Compares column vector y with each column of matrix x
(c) Compares row vector y with each row of matrix x

The output `number` is a scalar or vector that indicates the number of elements that differ. The size of `number` is determined by the optional input *`flg`* and by the dimensions of `x` and `y`. The output `ratio` equals `number` divided by the total number of elements in the *smaller* input.

For Specific Syntaxes

`[number,ratio] = symerr(x,y)` compares the elements in `x` and `y`. The sizes of `x` and `y` determine which elements are compared:

- If `x` and `y` are matrices of the same dimensions, then `symerr` compares `x` and `y` element by element. `number` is a scalar. See schematic (a) in the figure.
- If one is a row (respectively, column) vector and the other is a two-dimensional matrix, then `symerr` compares the vector element by element with *each row (resp., column)* of the matrix. The length

of the vector must equal the number of columns (resp., rows) in the matrix. `number` is a column (resp., row) vector whose mth entry indicates the number of elements that differ when comparing the vector with the mth row (resp., column) of the matrix. See schematics (b) and (c) in the figure.

`[number,ratio] = symerr(x,y,flg)` is similar to the previous syntax, except that *`flg`* can override the defaults that govern which elements `symerr` compares and how `symerr` computes the outputs. The values of *`flg`* are `'overall'`, `'column-wise'`, and `'row-wise'`. The table below describes the differences that result from various combinations of inputs. In all cases, `ratio` is `number` divided by the total number of elements in `y`.

Comparing a Two-Dimensional Matrix x with Another Input y

Shape of y	flg	Type of Comparison	number
Two-dim. matrix	`'overall'` (default)	Element by element	Total number of symbol errors
	`'column-wise'`	mth column of `x` vs. mth column of `y`	Row vector whose entries count symbol errors in each column
	`'row-wise'`	mth row of `x` vs. mth row of `y`	Column vector whose entries count symbol errors in each row

Comparing a Two-Dimensional Matrix x with Another Input y (Continued)

Shape of y	flg	Type of Comparison	number
Column vector	`'overall'`	`y` vs. each column of `x`	Total number of symbol errors
	`'column-wise'` (default)	`y` vs. each column of `x`	Row vector whose entries count symbol errors in each column of `x`
Row vector	`'overall'`	`y` vs. each row of `x`	Total number of symbol errors
	`'row-wise'` (default)	`y` vs. each row of `x`	Column vector whose entries count symbol errors in each row of `x`

`[number,ratio,loc] = symerr(...)` returns a binary matrix `loc` that indicates which elements of `x` and `y` differ. An element of `loc` is zero if the corresponding comparison yields no discrepancy, and one otherwise.

Examples

On the reference page for `biterr`, the last example uses `symerr`.

The command below illustrates how `symerr` works when one argument is a vector and the other is a matrix. It compares the vector `[1,2,3]'` to the columns

$$\begin{bmatrix}1\\3\\3\end{bmatrix}, \begin{bmatrix}1\\2\\3\end{bmatrix}, \begin{bmatrix}3\\2\\8\end{bmatrix}, \text{and } \begin{bmatrix}1\\2\\3\end{bmatrix}$$

of the matrix.

```
num = symerr([1 2 3]',[1 1 3 1;3 2 2 2; 3 3 8 3])

num =

     1     0     2     0
```

As another example, the command below illustrates the use of *`flg`* to override the default row-by-row comparison. Notice that `number` and `ratio` are scalars.

```
format rat;
[number,ratio,loc] = symerr([1 2; 3 4],[1 3],'overall')
```

The output is below.

```
number =

      3


ratio =

     3/4


loc =

      0            1
      1            1
```

See Also `biterr`

Purpose

Produce syndrome decoding table

Syntax

```
t = syndtable(h)
```

Description

`t = syndtable(h)` returns a decoding table for an error-correcting binary code having codeword length n and message length k. `h` is an (n-k)-by-n parity-check matrix for the code. `t` is a 2^{n-k}-by-n binary matrix. The rth row of `t` is an error pattern for a received binary codeword whose syndrome has decimal integer value r-1. (The syndrome of a received codeword is its product with the transpose of the parity-check matrix.) In other words, the rows of `t` represent the coset leaders from the code's standard array.

When converting between binary and decimal values, the leftmost column is interpreted as the *most* significant digit. This differs from the default convention in the `bi2de` and `de2bi` commands.

Examples

An example is in "Decoding Table".

References

[1] Clark, George C., Jr., and J. Bibb Cain, *Error-Correction Coding for Digital Communications*, New York, Plenum, 1981.

See Also

`decode` | `hammgen` | `gfcosets`

How To

- "Block Codes"

testconsole.Results

Purpose

Gets results from test console simulations

Description

The getResults method of the Error Rate Test Console returns an instance of a testconsole.Results object containing simulation results data. You use methods of the results object to retrieve and plot simulations results data.

Properties

A testconsole.Results object has the properties shown on the following table. All properties are writable except for the ones explicitly noted otherwise.

Property	Description
TestConsoleName	Error Rate Test Console. This property is not writable.
System Under Test Name	Name of the system under test for which the Error Rate Test Console obtained results. This property is not writable.
IterationMode	Iteration mode the Error Rate Test Console used for obtaining results. This property is not writable.
TestPoint	Specify the name of the registered test point for which the results object parses results. The getData, plot, and semilogy methods of the Results object return data or create a plot for the test point that the TestPoint property specifies.

Property	Description
Metric	Specify the name of the test metric for which the results object parses results. The getData, plot, and semilogy methods of the Results object returns data or creates a plot for the metric that the Metric property specifies.
TestParameter1	Specifies the name of the first independent variable for which the results object parses results.
TestParameter2	Specifies the name of the second independent variable for which the results object parses results.

Methods

A testconsole.Results object has the following methods.

getData

`d = getData(r)` returns results data matrix, *d*, available in the results object *r*. The returned results correspond to the test point currently specified in the `TestPoint` property of *r*, and to the test metric currently specified in the `Metric` property of *r*.

If `IterationMode` is 'Combinatorial' then *d* is a matrix containing results for all the sweep values available in the test parameters specified in the `TestParameter1` and `TestParameter2` properties. The rows of the matrix correspond to results for all the sweep values available in `TestParameter1`. The columns of the matrix correspond to results for all sweep values available in `TestParameter2`. If more than two test parameters are registered to the Error Rate Test Console, *d* contains results corresponding to the first value in the sweep vector of all parameters that are not `TestParameter1` or `TestParameter2`.

If `IterationMode` is 'Indexed', then *d* is a vector of results corresponding to each indexed combination of all the test parameter values registered to the Error Rate Test Console.

plot

`plot(r)` creates a plot for the results available in the results object r. The plot corresponds to the test point and test metric, specified by the `TestPoint` and `Metric` properties of *r*

If `IterationMode` is 'Combinatorial' then the plot contains a set of curves. The sweep values in TestParameter1 control the x-axis and the number of sweep values for TestParameter2 specifies how many curves the plot contains. If more than two test parameters are registered to the Error Rate Test Console, the curves correspond to results obtained with the first value in the sweep vector of all parameters that are not TestParameter1, or TestParameter2.

No plots are available when 'IterationMode' is 'Indexed'.

semilogy

`semilogy(...)` is the same as `plot(...)`, except that the Y-Axis uses a logarithmic (base 10) scale.

surf

surf(r) creates a 3-D, color, surface plot for the results available in the `results` object, r. The surface plot corresponds to following items:

- The test point you specify using the `TestPoint` property of the `results` object
- The test metric currently you specify in the `Metric` property of the `results` object

You can specify parameter/value pairs for the `results` object, which establishes additional properties of the surface plot.

When you select 'Combinatorial' for the IterationMode, the sweep values available in the test parameter you specify for the TestParameter1 property control the x-axis of the surface plot. The sweep values available in the test parameter you specify for the TestParameter2 property control the y-axis.

If more than two test parameters are registered to the test console, the surface plot corresponds to the results obtained with the parameter sweep values previously specified with the setParsingValues method of the results object.

You display the current parsing values by calling the getParsingValues method of the results object. The parsing values default to the first value in the sweep vector of each test parameter. By default, the surf method ignores the parsing values for any parameters currently set as TestParameter1 or TestParameter2.

No surface plots are available if the IterationMode is 'Indexed', when less than two registered test parameters exist, or TestParameter2 is set to 'None'.

setParsingValues

`setParsingValues(R,'ParameterName1', 'Value1', ... 'ParameterName2', 'Value2', ...)` sets the parsing values to the values you specify using the parameter-value pairs. Parameter name inputs must correspond to names of registered test parameters, and value inputs must correspond to a valid test parameter sweep value.

You use this method for specifying single sweep values for test parameters that differ from the values for TestParameter1 and TestParameter2. When you define this method, the `results` object returns the data values or plots corresponding to the sweep values you set for the setParsingValues method. The parsing values default to the first value in the sweep vector of each test parameter.

You display the current parsing values by calling the `getParsingValues` method of the results object. You may set parsing values for parameters in TestParameter1 and TestParameter2, but the results object ignores the values when getting data or returning plots.

Parsing values are irrelevant when `IterationMode` is 'Indexed'.

getParsingValues

`getParsingValues` displays the current parsing values for the Error Rate Test Console.

`s = getParsingValues(r)` returns a structure, s, with field names equal to the registered test parameter names and with values corresponding to the current parsing values.

Parsing values are irrelevant when IterationMode is 'Indexed'.

See Also `commtest.ErrorRate`

Purpose

Construct variable-step-size least mean square (LMS) adaptive algorithm object

Syntax

```
alg = varlms(initstep,incstep,minstep,maxstep)
```

Description

The `varlms` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see "Adaptive Algorithms".

`alg = varlms(initstep,incstep,minstep,maxstep)` constructs an adaptive algorithm object based on the variable-step-size least mean square (LMS) algorithm. `initstep` is the initial value of the step size parameter. `incstep` is the increment by which the step size changes from iteration to iteration. `minstep` and `maxstep` are the limits between which the step size can vary.

Properties

The table below describes the properties of the variable-step-size LMS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see "Access Properties of an Adaptive Algorithm".

Property	Description
`AlgType`	Fixed value, `'Variable Step Size LMS'`
`LeakageFactor`	LMS leakage factor, a real number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, while a value of 0 corresponds to a memoryless update algorithm.
`InitStep`	Initial value of step size when the algorithm starts

Property	Description
`IncStep`	Increment by which the step size changes from iteration to iteration
`MinStep`	Minimum value of step size
`MaxStep`	Maximum value of step size

Also, when you use this adaptive algorithm object to create an equalizer object (via the `lineareq` or `dfe` function), the equalizer object has a `StepSize` property. The property value is a vector that lists the current step size for each weight in the equalizer.

Examples

For an example that uses this function, see "Linked Properties of an Equalizer Object".

Algorithms

Referring to the schematics presented in "Equalizer Structure", define w as the vector of all current weights w_i and define u as the vector of all inputs u_i. Based on the current step size, μ, this adaptive algorithm first computes the quantity

$$\mu_0 = \mu + (\texttt{IncStep})\ \mathrm{Re}(g g_{prev})$$

where $g = ue^*$, g_{prev} is the analogous expression from the previous iteration, and the * operator denotes the complex conjugate.

Then the new step size is given by

- μ_0, if it is between `MinStep` and `MaxStep`
- `MinStep`, if μ_0 < `MinStep`
- `MaxStep`, if μ_0 > `MaxStep`

The new set of weights is given by

$$(\texttt{LeakageFactor})\ w + 2\ \mu\ g^*$$

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

See Also

`lms` | `signlms` | `normlms` | `rls` | `cma` | `lineareq` | `dfe` | `equalize`

How To

- "Equalization"

vec2mat

Purpose

Convert vector into matrix

Syntax

```
mat = vec2mat(vec,matcol)
mat = vec2mat(vec,matcol,padding)
[mat,padded] = vec2mat(...)
```

Description

`mat = vec2mat(vec,matcol)` converts the vector `vec` into a matrix with `matcol` columns, creating one row at a time. If the length of `vec` is not a multiple of `matcol`, then extra zeros are placed in the last row of `mat`. The matrix `mat` has `ceil(length(vec)/matcol)` rows.

`mat = vec2mat(vec,matcol,padding)` is the same as the first syntax, except that the extra entries placed in the last row of `mat` are not necessarily zeros. The extra entries are taken from the matrix `padding`, in order. If `padding` has fewer entries than are needed, then the last entry is used repeatedly.

`[mat,padded] = vec2mat(...)` returns an integer `padded` that indicates how many extra entries were placed in the last row of `mat`.

Note `vec2mat` is similar to the built-in MATLAB function `reshape`. However, given a vector input, `reshape` creates a matrix one *column* at a time instead of one row at a time. Also, `reshape` requires the input and output matrices to have the same number of entries, whereas `vec2mat` places extra entries in the output matrix if necessary.

Examples

```
vec = [1 2 3 4 5];
[mat,padded] = vec2mat(vec,3)
[mat2,padded2] = vec2mat(vec,4)
mat3 = vec2mat(vec,4,[10 9 8; 7 6 5; 4 3 2])
```

The output is below.

```
mat =

     1     2     3
```

```
     4     5     0


padded =

     1


mat2 =

     1     2     3     4
     5     0     0     0


padded2 =

     3


mat3 =

     1     2     3     4
     5    10     7     4
```

See Also

`reshape`

Purpose

Convolutionally decode binary data using Viterbi algorithm

Syntax

```
decoded = vitdec(code,trellis,tblen,opmode,dectype)
decoded = vitdec(code,trellis,tblen,opmode,'soft',nsdec)
decoded = ...
  vitdec(code,trellis,tblen,opmode,dectype,puncpat)
decoded = ...
  vitdec(code,trellis,tblen,opmode,dectype,puncpat,eraspat)
decoded = ...
  vitdec(...,'cont',...,initmetric,initstates,initinputs)
[decoded,finalmetric,finalstates,finalinputs] = ...
  vitdec(...,'cont',...)
```

Description

`decoded = vitdec(code,trellis,tblen,`*`opmode`*`,`*`dectype`*`)` decodes the vector `code` using the Viterbi algorithm. The MATLAB structure `trellis` specifies the convolutional encoder that produced `code`; the format of `trellis` is described in "Trellis Description of a Convolutional Code" and the reference page for the `istrellis` function. `code` contains one or more symbols, each of which consists of `log2(trellis.numOutputSymbols)` bits. Each symbol in the vector `decoded` consists of `log2(trellis.numInputSymbols)` bits. `tblen` is a positive integer scalar that specifies the traceback depth. If the code rate is 1/2, a typical value for `tblen` is about five times the constraint length of the code.

The string *`opmode`* indicates the decoder's operation mode and its assumptions about the corresponding encoder's operation. Choices are in the table below.

Values of opmode Input

Value	Meaning
`'cont'`	The encoder is assumed to have started at the all-zeros state. The decoder traces back from the state with the best metric. A delay equal to `tblen` symbols elapses before the first decoded symbol appears in the output. This mode is appropriate when you invoke this function repeatedly and want to preserve continuity between successive invocations. See the continuous operation mode syntaxes below.
`'term'`	The encoder is assumed to have both started and ended at the all-zeros state, which is true for the default syntax of the `convenc` function. The decoder traces back from the all-zeros state. This mode incurs no delay. This mode is appropriate when the uncoded message (that is, the input to `convenc`) has enough zeros at the end to fill all memory registers of the encoder. If the encoder has `k` input streams and constraint length vector `constr` (using the polynomial description of the encoder), "enough" means `k*max(constr-1)`.
`'trunc'`	The encoder is assumed to have started at the all-zeros state. The decoder traces back from the state with the best metric. This mode incurs no delay. This mode is appropriate when you cannot assume the encoder ended at the all-zeros state and when you do not want to preserve continuity between successive invocations of this function.

For the `'term'` and `'trunc'` mode, the traceback depth (`tblen`) must be a positive integer scalar value, not greater than the number of input symbols in `code`.

The string *`dectype`* indicates the type of decision that the decoder makes, and influences the type of data the decoder expects in `code`. Choices are in the table below.

Values of dectype Input

Value	Meaning
'unquant'	code contains real input values, where 1 represents a logical zero and -1 represents a logical one.
'hard'	code contains binary input values.
'soft'	For soft-decision decoding, use the syntax below. nsdec is required for soft-decision decoding.

Syntax for Soft Decision Decoding

`decoded = vitdec(code,trellis,tblen,`*`opmode`*`,'soft',nsdec)` decodes the vector `code` using soft-decision decoding. `code` consists of integers between 0 and `2^nsdec-1`, where 0 represents the most confident 0 and `2^nsdec-1` represents the most confident 1. The existing implementation of the functionality supports up to 13 bits of quantization, meaning `nsdec` can be set up to 13. For reference, 3 bits of quantization is about 2 db better than hard decision decoding.

Syntax for Punctures and Erasures

`decoded = ...`
`vitdec(code,trellis,tblen,`*`opmode`*`,`*`dectype`*`,puncpat)` denotes the input punctured `code`, where `puncpat` is the puncture pattern vector, and where `0`s indicate punctured bits in the input code.

`decoded = ...`
`vitdec(code,trellis,tblen,`*`opmode`*`,`*`dectype`*`,puncpat,eraspat)` allows an erasure pattern vector, `eraspat`, to be specified for the input `code`, where the `1`s indicate the corresponding erasures. `eraspat` and `code` must be of the same length. If puncturing is not used, specify `puncpat` to be `[]`. In the `eraspat` vector, `1`s indicate erasures in the input code.

Additional Syntaxes for Continuous Operation Mode

Continuous operation mode enables you to save the decoder's internal state information for use in a subsequent invocation of this function. Repeated calls to this function are useful if your data is partitioned into a series of smaller vectors that you process within a loop, for example.

`decoded = ... vitdec(...,'cont',...,initmetric,initstates,initinputs)` is the same as the earlier syntaxes, except that the decoder starts with its state metrics, traceback states, and traceback inputs specified by `initmetric`, `initstates`, and `initinputs`, respectively. Each real number in `initmetric` represents the starting state metric of the corresponding state. `initstates` and `initinputs` jointly specify the initial traceback memory of the decoder; both are `trellis.numStates`-by-`tblen` matrices. `initstates` consists of integers between 0 and `trellis.numStates-1`. If the encoder schematic has more than one input stream, the shift register that receives the first input stream provides the least significant bits in `initstates`, while the shift register that receives the last input stream provides the most significant bits in `initstates`. The vector `initinputs` consists of integers between 0 and `trellis.numInputSymbols-1`. To use default values for all of the last three arguments, specify them as `[],[],[]`.

`[decoded,finalmetric,finalstates,finalinputs] = ... vitdec(...,'cont',...)` is the same as the earlier syntaxes, except that the final three output arguments return the state metrics, traceback states, and traceback inputs, respectively, at the end of the decoding process. `finalmetric` is a vector with `trellis.numStates` elements that correspond to the final state metrics. `finalstates` and `finalinputs` are both matrices of size `trellis.numStates`-by-`tblen`. The elements of `finalstates` have the same format as those of `initstates`.

Traceback Matrices

The t^{th} column of P_1 shows the t-1^{th} time step states given the inputs listed in the input matrix. For example, the value in the i^{th} row shows the state at time t-1 that transitions to the i-1 state at time t. The input required for this state transition is given in the i^{th} row of the t^{th} column of the input matrix.

The P_1 output is the states of the traceback matrix. It is a [number of states x traceback length] matrix. The following example uses a (7,5), rate 1/2 code. This code is easy to follow:

t = poly2trellis(3,[7 5]);
k = log2(t.numInputSymbols);
msg = [1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0];
code = convenc(msg,t); tblen = 15; [d1 m1 p1
in1]=vitdec(code(1:end/2),t,tblen,'cont','hard')

```
m1 =

     0     3     2     3

p1 =

     0     1     1     0     0     1     1     0     0     1     1     0
     2     3     3     2     2     3     3     2     2     3     3     2
     0     1     1     0     0     1     1     0     0     1     1     0
     2     3     3     2     2     3     3     2     2     3     3     2

in1 =

     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     1     1     1     1     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1     1     1     1     1
```

In this example, the message makes the encoder states follow the following sequence:

0 2 3 1 / 0 2 3 1 / ...

Since the best state is `0` (column index of smallest metric in m_1 –1), the traceback matrix starts from sate `0`, looking at the first row (`0`th state) of the last column of P_1, ([1; 3; 1; 3]), which is 1. This indicates 1 for the previous state.

Next, the traceback matrix checks in1 ([0; 0; 1; 1]), which indicates 0 for the input. The second row (1st state) of the 14th column of P_1 ([1; 3; 1; 3]) is 3. This indicates 3 for the previous state.

The traceback matrix checks in1 ([0; 0; 1; 1]), which indicates that the input was 0. The fourth row (3rd state) of the 13th column of P_1 ([0; 2; 0; 2]), is 2. This indicates 2 for the previous state.

The traceback matrix checks in1 ([0; 0; 1; 1]), which indicates the input was 1. The third row (2nd state) of the 12th column of P_1 ([0; 2; 0; 2]), is 0. This indicates 0 for the previous state.

The traceback matrix checks in1 ([0; 0; 1; 1]), which indicates the input was 1. The first row (0th state) of the 11th column of P_1 ([1; 3; 1; 3]), is 1. This indicates 1 for the previous state. Then, the matrix checks in1 ([0; 0; 1; 1]), which indicates 0 for the input.

To determine the best state for a gicen time, use m_1. The smallest number in m_1 represents the best state.

Examples

The example below encodes random data and adds noise. Then it decodes the noisy code three times to illustrate the three decision types that `vitdec` supports. For unquantized and soft decisions, the output of `convenc` does not have the same data type that `vitdec` expects for the input code, so it is necessary to manipulate `ncode` before invoking `vitdec`. That the bit error rate computations must account for the delay that the continuous operation mode incurs.

```
s = RandStream.create('mt19937ar', 'seed',131);
prevStream = RandStream.setGlobalStream(s); % seed for repeatability
trel = poly2trellis(3,[6 7]); % Define trellis.
msg = randi([0 1],100,1); % Random data
code = convenc(msg,trel); % Encode.
ncode = rem(code + randerr(200,1,[0 1;.95 .05]),2); % Add noise.
tblen = 3; % Traceback length
decoded1 = vitdec(ncode,trel,tblen,'cont','hard'); %Hard decision
```

```
% Use unquantized decisions.
ucode = 1-2*ncode; % +1 & -1 represent zero & one, respectively.
decoded2 = vitdec(ucode,trel,tblen,'cont','unquant');
% To prepare for soft-decision decoding, map to decision values.
[x,qcode] = quantiz(1-2*ncode,[-.75 -.5 -.25 0 .25 .5 .75],...
[7 6 5 4 3 2 1 0]); % Values in qcode are between 0 and 2^3-1.
decoded3 = vitdec(qcode',trel,tblen,'cont','soft',3);
% Compute bit error rates, using the fact that the decoder
% output is delayed by tblen symbols.
[n1,r1] = biterr(decoded1(tblen+1:end),msg(1:end-tblen));
[n2,r2] = biterr(decoded2(tblen+1:end),msg(1:end-tblen));
[n3,r3] = biterr(decoded3(tblen+1:end),msg(1:end-tblen));
disp(['The bit error rates are:   ',num2str([r1 r2 r3])])
RandStream.setGlobalStream(prevStream); % restore default stream
```

The following example illustrates how to use the final state and initial state arguments when invoking `vitdec` repeatedly. `[decoded4;decoded5]` is the same as `decoded6`.

```
s = RandStream.create('mt19937ar', 'seed',131); % seed for repeatability
prevStream = RandStream.setGlobalStream(s);
trel = poly2trellis(3,[6 7]);
code = convenc(randi([0 1],100,1),trel);
% Decode part of code, recording final state for later use.
[decoded4,f1,f2,f3] = vitdec(code(1:100),trel,3,'cont','hard');
% Decode the rest of code, using state input arguments.
decoded5 = vitdec(code(101:200),trel,3,'cont','hard',f1,f2,f3);
% Decode the entire code in one step.
decoded6 = vitdec(code,trel,3,'cont','hard');
isequal(decoded6,[decoded4; decoded5])
RandStream.setGlobalStream(prevStream); % restore default stream
```

For additional examples, see “Convolutional Codes”.

For some commonly used puncture patterns for specific rates and polynomials, see the last three references below.

References

[1] Clark, G. C. Jr. and J. Bibb Cain., *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

[2] Gitlin, Richard D., Jeremiah F. Hayes, and Stephen B. Weinstein, *Data Communications Principles*, New York, Plenum, 1992.

[3] Heller, J. A. and I. M. Jacobs, "Viterbi Decoding for Satellite and Space Communication," *IEEE Transactions on Communication Technology*, Vol. COM-19, October 1971, pp 835–848.

[4] Yasuda, Y., et. al., "High rate punctured convolutional codes for soft decision Viterbi decoding," *IEEE Transactions on Communications*, vol. COM-32, No. 3, pp 315–319, Mar. 1984.

[5] Haccoun, D., and G. Begin, "High-rate punctured convolutional codes for Viterbi and sequential decoding," *IEEE Transactions on Communications*, vol. 37, No. 11, pp 1113–1125, Nov. 1989.

[6] G. Begin, et.al., "Further results on high-rate punctured convolutional codes for Viterbi and sequential decoding," *IEEE Transactions on Communications*, vol. 38, No. 11, pp 1922–1928, Nov. 1990.

See Also

`convenc` | `poly2trellis` | `istrellis`

How To

- viterbisim
- "Convolutional Codes"

Purpose Generate white Gaussian noise

Syntax

```
y = wgn(m,n,p)
y = wgn(m,n,p,imp)
y = wgn(m,n,p,imp,state)
y = wgn(...,powertype)
y = wgn(...,outputtype)
```

Description

`y = wgn(m,n,p)` generates an `m`-by-`n` matrix of white Gaussian noise. `p` specifies the power of `y` in decibels relative to a watt. The default load impedance is 1 ohm.

`y = wgn(m,n,p,imp)` is the same as the previous syntax, except that `imp` specifies the load impedance in ohms.

y = wgn(m,n,p,imp,s) uses *s*, which is a random stream handle, to generate random noise samples with randn. This syntax is useful to generate repeatable outputs. Type `help RandStream` for more information.

`y = wgn(m,n,p,imp,state)` is the same as the previous syntax, except that `wgn` first resets the state of the normal random number generator `randn` to the integer `state`.

Note This usage is deprecated and may be removed in a future release. Instead of `state`, use `s`, as in the previous example.

`y = wgn(...,`*`powertype`*`)` is the same as the previous syntaxes, except that the string *`powertype`* specifies the units of `p`. Choices for *`powertype`* are `'dBW'`, `'dBm'`, and `'linear'`.

`y = wgn(...,`*`outputtype`*`)` is the same as the previous syntaxes, except that the string *`outputtype`* specifies whether the noise is real or complex. Choices for *`outputtype`* are `'real'` and `'complex'`. If *`outputtype`* is `'complex'`, then the real and imaginary parts of `y` each have a noise power of `p`/2.

Note The unit of measure for the output of the `wgn` function is Volts. For power calculations, it is assumed that there is a load of 1 Ohm.

Examples

To generate a column vector of length 100 containing real white Gaussian noise of power 0 dBW, use this command:

```
y1 = wgn(100,1,0);
```

To generate a column vector of length 100 containing complex white Gaussian noise, each component of which has a noise power of 0 dBW, use this command:

```
y2 = wgn(100,1,0,'complex');
```

See Also

`randn` | `awgn`

How To

- “Sources and Sinks”

2

Blocks — Alphabetical List

A-Law Compressor

Purpose

Implement A-law compressor for source coding

Library

Source Coding

Description

The A-Law Compressor block implements an A-law compressor for the input signal. The formula for the A-law compressor is

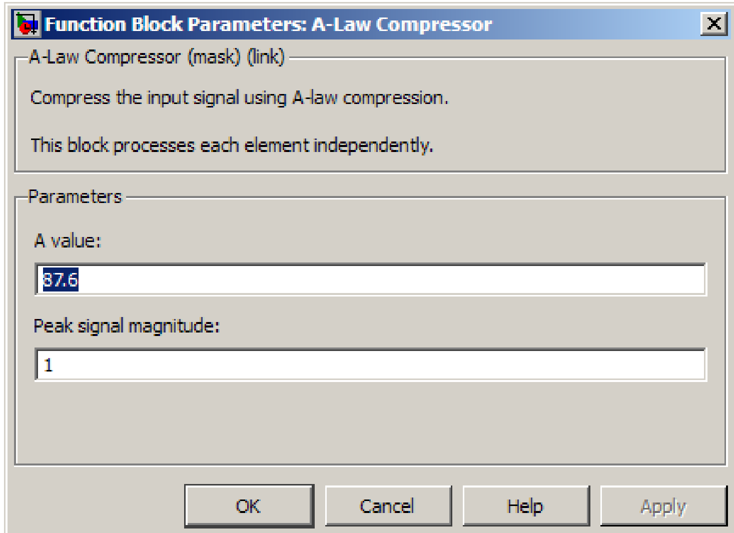
$$y = \begin{cases} \dfrac{A|x|}{1+\log A}\,\mathrm{sgn}(x) & \text{for } 0 \le |x| \le \dfrac{V}{A} \\[2ex] \dfrac{V\left(1+\log(A|x|/V)\right)}{1+\log A}\,\mathrm{sgn}(x) & \text{for } \dfrac{V}{A} < |x| \le V \end{cases}$$

where A is the A-law parameter of the compressor, V is the peak signal magnitude for x, log is the natural logarithm, and sgn is the signum function (`sign` in MATLAB software).

The most commonly used A value is 87.6.

The input can have any shape or frame status. This block processes each vector element independently.

Dialog Box



A value
The A-law parameter of the compressor.

Peak signal magnitude
The peak value of the input signal. This is also the peak value of the output signal.

Supported Data Type

Port	Supported Data Types
In	• double
Out	• double

Pair Block

A-Law Expander

See Also

Mu-Law Compressor

References

[1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, N.J., Prentice-Hall, 1988.

A-Law Expander

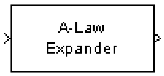
Purpose

Implement A-law expander for source coding

Library

Source Coding

Description

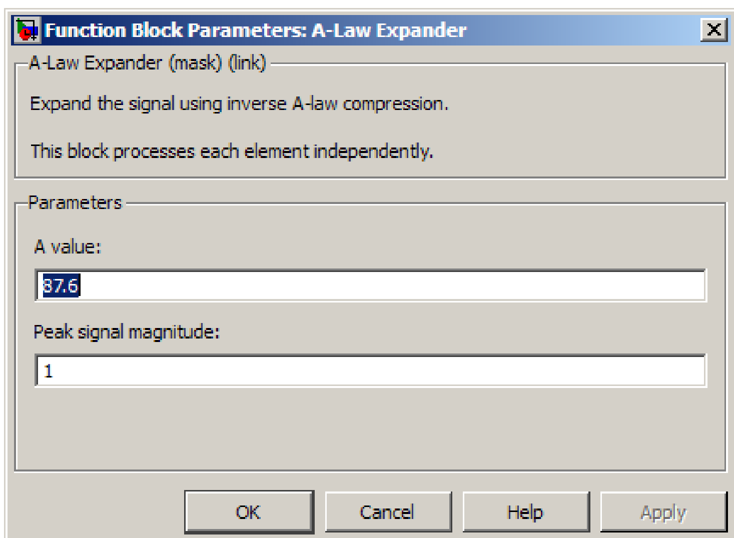


The A-Law Expander block recovers data that the A-Law Compressor block compressed. The formula for the A-law expander, shown below, is the inverse of the compressor function.

$$x = \begin{cases} \dfrac{y(1+\log A)}{\mathrm{A}} & \text{for } 0 \le |y| \le \dfrac{\mathrm{V}}{1+\log\mathrm{A}} \\[2ex] \exp\left(|\mathrm{y}|(1+\log A)/V - 1\right)\dfrac{V}{A}\operatorname{sgn}(y) & \text{for } \dfrac{\mathrm{V}}{1+\log\mathrm{A}} < |y| \le V \end{cases}$$

The input can have any shape or frame status. This block processes each vector element independently.

Dialog Box



A value

The A-law parameter of the compressor.

Peak signal magnitude
The peak value of the input signal. This is also the peak value of the output signal.

Match these parameters to the ones in the corresponding A-Law Compressor block.

Supported Data Type

Port	Supported Data Types
In	• double
Out	• double

Pair Block

A-Law Compressor

See Also

Mu-Law Expander

References

[1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, N.J., Prentice-Hall, 1988.

Algebraic Deinterleaver

Purpose

Restore ordering of input symbols using algebraically derived permutation

Library

Block sublibrary of Interleaving

Description

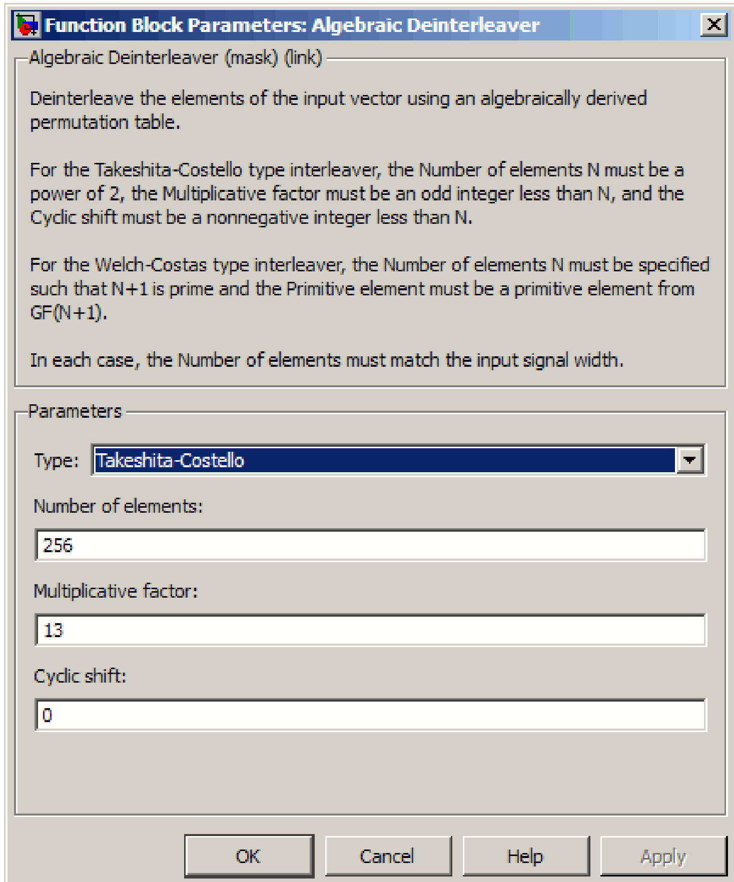
The Algebraic Deinterleaver block restores the original ordering of a sequence that was interleaved using the Algebraic Interleaver block. In typical usage, the parameters in the two blocks have the same values.

The **Number of elements** parameter, *N*, indicates how many numbers are in the input vector. This block accepts a column vector input signal.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

The **Type** parameter indicates the algebraic method that the block uses to generate the appropriate permutation table. Choices are `Takeshita-Costello` and `Welch-Costas`. Each of these methods has parameters and restrictions that are specific to it; these are described on the reference page for the Algebraic Interleaver block.

Dialog Box



Type

The type of permutation table that the block uses for deinterleaving. Choices are `Takeshita-Costello` and `Welch-Costas`.

Number of elements

The number of elements, N, in the input vector.

Multiplicative factor
The factor the block uses to compute the corresponding interleaver's cycle vector. This field appears only when you set **Type** to `Takeshita-Costello`.

Cyclic shift
The amount by which the block shifts indices when creating the corresponding interleaver's permutation table. This field appears only when you set **Type** to `Takeshita-Costello`.

Primitive element
An element of order N in the finite field GF(N+1). This field appears only if **Type** is set to `Welch-Costas`.

Pair Block

Algebraic Interleaver

See Also

General Block Deinterleaver

References

[1] Heegard, Chris and Stephen B. Wicker. *Turbo Coding*. Boston: Kluwer Academic Publishers, 1999.

[2] Takeshita, O. Y. and D. J. Costello, Jr. "New Classes Of Algebraic Interleavers for Turbo-Codes." *Proc. 1998 IEEE International Symposium on Information Theory*, Boston, Aug. 16-21, 1998. 419.

Purpose

Reorder input symbols using algebraically derived permutation table

Library

Block sublibrary of Interleaving

Description

The Algebraic Interleaver block rearranges the elements of its input vector using a permutation that is algebraically derived. The **Number of elements** parameter, N, indicates how many numbers are in the input vector. This block accepts a column vector input signal.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

The **Type** parameter indicates the algebraic method that the block uses to generate the appropriate permutation table. Choices are `Takeshita-Costello` and `Welch-Costas`. Each of these methods has parameters and restrictions that are specific to it:

- If you set **Type** to `Welch-Costas`, then $N + 1$ must be prime. The **Primitive element** parameter is an integer, A, between 1 and N that represents a primitive element of the finite field GF($N + 1$). This means that every nonzero element of GF($N + 1$) can be expressed as A raised to some integer power.

  In a Welch-Costas interleaver, the permutation maps the integer k to mod(A^k,$N + 1$) - 1.

- If you set **Type** to `Takeshita-Costello`, then N must be 2^m for some integer m. The **Multiplicative factor** parameter, k, must be an odd integer less than N. The **Cyclic shift** parameter, h, must be a nonnegative integer less than N.

  A Takeshita-Costello interleaver uses a length-N *cycle vector* whose n^{th} element is

$$c(n) = \mathrm{mod}\left(k \cdot \frac{n \cdot (n-1)}{2}, N\right) + 1, n$$

for integers n between 1 and N. The intermediate permutation function is obtained by using the following relationship:

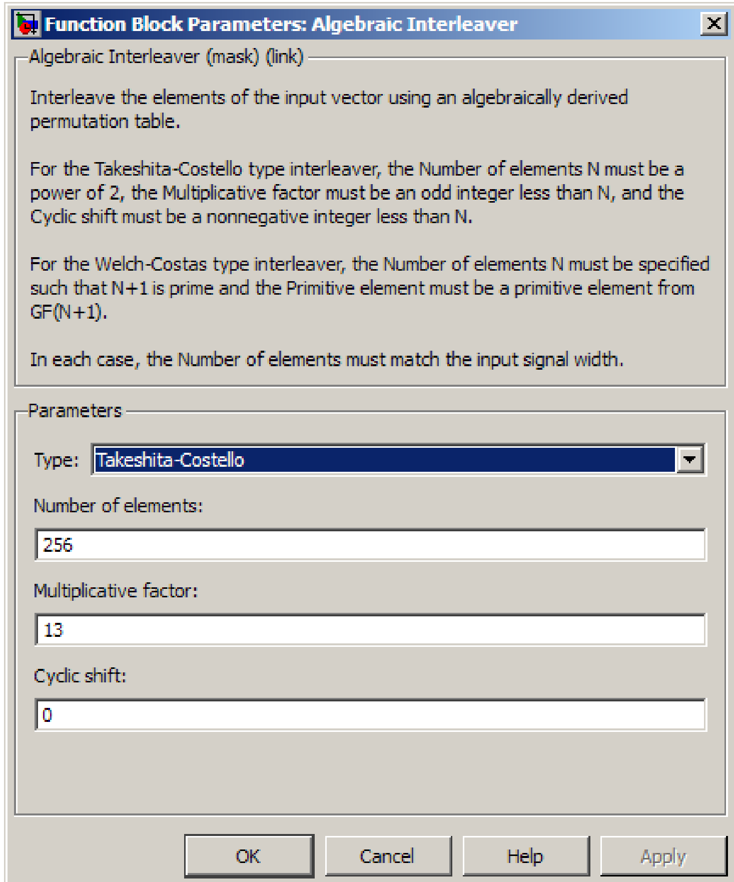
$$\Pi\left(c(n)\right) = c(n+1)$$

where

$$n = 1 : N$$

The interleaver's actual permutation vector is the result of cyclically shifting the elements of the permutation vector, π, by the **Cyclic shift** parameter, *h*.

Dialog Box



Type

The type of permutation table that the block uses for interleaving.

Number of elements

The number of elements, N, in the input vector.

Multiplicative factor

The factor used to compute the interleaver's cycle vector. This field appears only if **Type** is set to `Takeshita-Costello`.

Cyclic shift
The amount by which the block shifts indices when creating the permutation table. This field appears only if **Type** is set to `Takeshita-Costello`.

Primitive element
An element of order *N* in the finite field GF(N+1). This field appears only if **Type** is set to `Welch-Costas`.

Pair Block

Algebraic Deinterleaver

See Also

General Block Interleaver

References

[1] Heegard, Chris and Stephen B. Wicker. *Turbo Coding*. Boston: Kluwer Academic Publishers, 1999.

[2] Takeshita, O. Y. and D. J. Costello, Jr. "New Classes Of Algebraic Interleavers for Turbo-Codes." *Proc. 1998 IEEE International Symposium on Information Theory*, Boston, Aug. 16-21, 1998. 419.

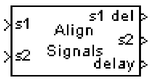
Purpose

Align two signals by finding delay between them

Library

Utility Blocks

Description



Align Signals

The Align Signals block aligns two signals by finding the delay between them. This is useful when you want to compare a transmitted and received signal to determine the bit error rate, but do not know the delay in the received signal. This block accepts a column vector or matrix input signal. For a matrix input, the block aligns each channel independently.

The `s1` input port receives the original signal, while the `s2` input port receives a delayed version. The two input signals must have the same dimensions and sample times. The block calculates the delay between the two signals, and then

- Delays the first signal, `s1`, by the calculated value, and outputs it through the port labeled `s1 del`.
- Outputs the second signal `s2` without change through the port labeled `s2`.
- Outputs the delay value through the port labeled `delay`.

See "Delays" in the Communications System Toolbox online documentation for more information about signal delays.

The block's **Correlation window length** parameter specifies how many samples of the signals the block uses to calculate the cross-correlation. The delay output is a nonnegative integer less than the **Correlation window length**.

As the **Correlation window length** is increased, the reliability of the computed delay also increases. However, the processing time to compute the delay increases as well.

You can make the Align Signals block stop updating the delay after it computes the same delay value for a specified number of samples. To do so, select **Disable recurring updates**, and enter a positive integer in the **Number of constant delay outputs to disable updates** field. For example, if you set **Number of constant delay outputs to disable updates** to 20, the block will stop recalculating and updating the delay after it calculates the same value 20 times in succession. Disabling recurring updates causes the simulation to run faster after the target number of constant delays occurs.

Tips for Using the Block Effectively

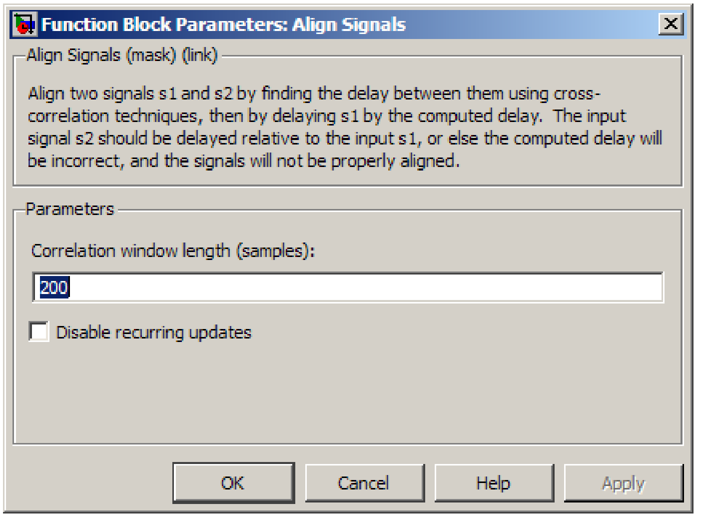
- Set the **Correlation window length** parameter sufficiently large so that the computed delay eventually stabilizes at a constant value. If the computed delay is not constant, you should increase **Correlation window length**. If the increased value of **Correlation window length** exceeds the duration of the simulation, then you should also increase the duration of the simulation accordingly.
- If the cross-correlation between the two signals is broad, then **Correlation window length** should be much larger than the expected delay, or else the algorithm might stabilize at an incorrect value. For example, a CPM signal has a broad autocorrelation, so it has a broad cross-correlation with a delayed version of itself. In this case, the **Correlation window length** value should be much larger than the expected delay.
- If the block calculates a delay that is greater than 75 percent of **Correlation window length**, the signal `s1` is probably delayed relative to the signal `s2`. In this case, you should switch the signal lines leading into the two input ports.
- If you use the Align Signals block with the Error Rate Calculation block, you should set the **Receive delay** parameter of the Error Rate Calculation block to `0` because the Align Signals block compensates for the delay. Also, you might want to set the Error Rate Calculation block's **Computation delay** parameter to a nonzero value to account for the possibility that the Align Signals block takes a nonzero amount of time to stabilize on the correct amount by which to delay one of the signals.

Examples

See the “Delays” section of *Communications System Toolbox User’s Guide* for an example that uses the Align Signals block in conjunction with the Error Rate Calculation block.

See Setting the Correlation Window Length, on the reference page for the Find Delay block, for an example that illustrates how to set the correlation window length properly.

Dialog Box



Correlation window length
: The number of samples the block uses to calculate the cross-correlations of the two signals.

Disable recurring updates
: Selecting this option causes the block to stop computing the delay after it computes the same delay value for a specified number of samples.

Number of constant delay outputs to disable updates
: A positive integer specifying how many times the block must compute the same delay before ceasing to update. This field appears only if **Disable recurring updates** is selected.

Algorithm

The Align Signals block finds the delay by calculating the cross-correlations of the first signal with time-shifted versions of the second signal, and then finding the index at which the cross-correlation is maximized.

See Also

Find Delay, Error Rate Calculation

APP Decoder

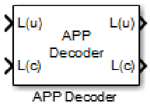
Purpose

Decode convolutional code using a posteriori probability (APP) method

Library

Convolutional sublibrary of Error Detection and Correction

Description



The APP Decoder block performs a posteriori probability (APP) decoding of a convolutional code.

Input Signals and Output Signals

The input $L(u)$ represents the sequence of log-likelihoods of encoder input bits, while the input $L(c)$ represents the sequence of log-likelihoods of code bits. The outputs $L(u)$ and $L(c)$ are updated versions of these sequences, based on information about the encoder.

If the convolutional code uses an alphabet of 2^n possible symbols, this block's $L(c)$ vectors have length $Q*n$ for some positive integer Q. Similarly, if the decoded data uses an alphabet of 2^k possible output symbols, then this block's L(u) vectors have length $Q*k$.

This block accepts a column vector input signal with any positive integer for Q.

If you only need the input $L(c)$ and output $L(u)$, you can attach a Simulink Ground block to the input $L(u)$ and a Simulink Terminator block to the output $L(c)$.

This block accepts `single` and `double` data types. Both inputs, however, must be of the same type. The output data type is the same as the input data type.

Specifying the Encoder

To define the convolutional encoder that produced the coded input, use the **Trellis structure** parameter. This parameter is a MATLAB structure whose format is described in "Trellis Description of a Convolutional Code" in the Communications System Toolbox documentation. You can use this parameter field in two ways:

- If you have a variable in the MATLAB workspace that contains the trellis structure, enter its name as the **Trellis structure** parameter. This way is preferable because it causes Simulink to spend less time updating the diagram at the beginning of each simulation, compared to the usage described next.

- If you want to specify the encoder using its constraint length, generator polynomials, and possibly feedback connection polynomials, use a `poly2trellis` command within the **Trellis structure** field. For example, to use an encoder with a constraint length of 7, code generator polynomials of 171 and 133 (in octal numbers), and a feedback connection of 171 (in octal), set the **Trellis structure** parameter to

  ```
  poly2trellis(7,[171 133],171)
  ```

To indicate how the encoder treats the trellis at the beginning and end of each frame, set the **Termination method** parameter to either `Truncated` or `Terminated`. The `Truncated` option indicates that the encoder resets to the all-zeros state at the beginning of each frame. The `Terminated` option indicates that the encoder forces the trellis to end each frame in the all-zeros state. If you use the Convolutional Encoder block with the **Operation mode** parameter set to `Truncated (reset every frame)`, use the `Truncated` option in this block. If you use the Convolutional Encoder block with the **Operation mode** parameter set to `Terminate trellis by appending bits`, use the `Terminated` option in this block.

Specifying Details of the Algorithm

You can control part of the decoding algorithm using the **Algorithm** parameter. The `True APP` option implements a posteriori probability decoding as per equations 20–23 in section V of [1]. To gain speed, both the `Max*` and `Max` options approximate expressions like

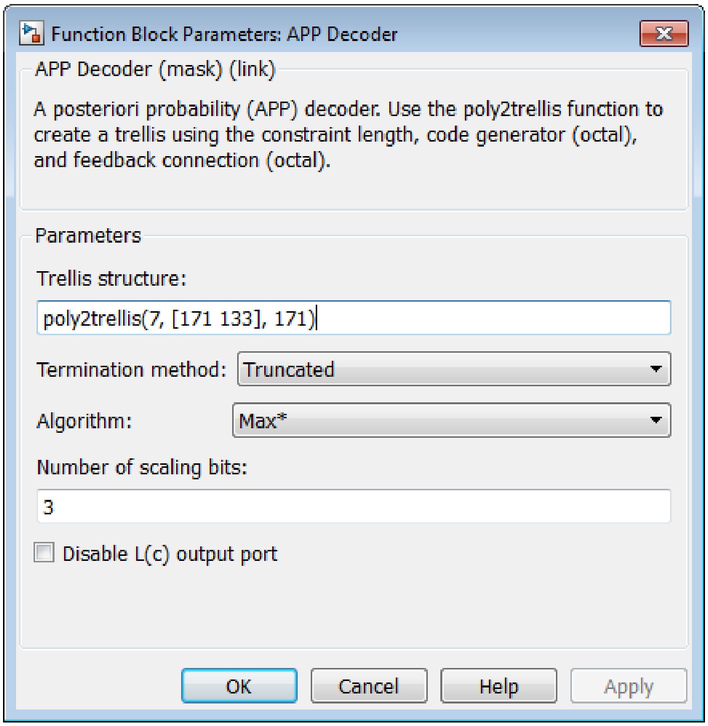
$$\log \sum_i \exp(a_i)$$

by other quantities. The `Max` option uses $\max(a_i)$ as the approximation, while the `Max*` option uses $\max(a_i)$ plus a correction term given by

$\ln(1+\exp(-|a_{i-1}-a_i|))$ [3].

The `Max*` option enables the **Scaling bits** parameter in the dialog box. This parameter is the number of bits by which the block scales the data it processes internally (multiplies the input by (`2^numScalingBits`) and divides the pre-output by the same factor). Use this parameter to avoid losing precision during the computations.

Dialog Box



Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Termination method

Either `Truncated` or `Terminated`. This parameter indicates how the convolutional encoder treats the trellis at the beginning and end of frames.

Algorithm

Either `True APP`, `Max*`, or `Max`.

Number of scaling bits

An integer between 0 and 8 that indicates by how many bits the decoder scales data in order to avoid losing precision. This field is active only when **Algorithm** is set to `Max*`.

Disable L(c) output port

Select this check box to disable the secondary block output, L(c).

Examples

For an example using this block, see the Iterative Decoding of a Serially Concatenated Convolutional Code example.

See Also

Viterbi Decoder, Convolutional Encoder;`poly2trellis`

References

[1] Benedetto, S., G. Montorsi, D. Divsalar, and F. Pollara, “A Soft-Input Soft-Output Maximum A Posterior (MAP) Module to Decode Parallel and Serial Concatenated Codes,” *JPL TDA Progress Report*, Vol. 42-127, November 1996.

[2] Benedetto, Sergio and Guido Montorsi, “Performance of Continuous and Blockwise Decoded Turbo Codes.” *IEEE Communications Letters*, Vol. 1, May 1997, 77–79.

[3] Viterbi, Andrew J., “An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes,” *IEEE Journal on Selected Areas in Communications*, Vol. 16, February 1998, 260–264.

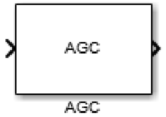
Purpose

Adaptively adjust gain for constant signal-level output

Library

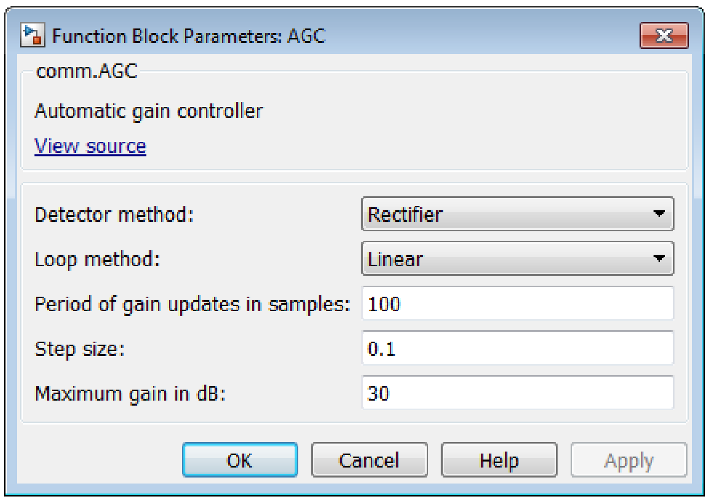
RF Impairments Correction

Description



This automatic gain controller (AGC) block adaptively adjusts its gain to achieve a constant signal level at the output.

Dialog Box



Detector method

Specify the method that the block uses to perform envelope detection. The default is `Rectifier`.

When you select `Rectifier`, the AGC detector outputs a voltage value proportional to the envelope amplitude of the output signal. The detector rectifies and then averages the input signal over the period of gain updates in samples. The AGC adjusts the gain to obtain unity voltage at the output of the detector.

When you select `Square law`, the AGC detector outputs a power value that is proportional to the square of the output voltage. The detector squares and then averages the input signal over the period of gain updates in samples. The AGC adjusts the gain to obtain unity power at the output of the detector.

Loop method

Specify the AGC loop implementation that the block uses. The default is `Linear`.

When you select `Linear`, the AGC uses the direct value of the detector output to determine the gain value. Typically, a linear loop responds quickly to increases in the input signal level. However, the loop's response to decreases in the input signal level tends to be slow.

When you select `Logarithmic`, the AGC uses the logarithm of the detector output to determine the gain value. Logarithmic loops respond to decreases in the input signal level much more quickly than linear loops.

Period of gain updates in samples

Specify the period of the gain updates as a double- or single-precision, real, integer-valued scalar. The default is `100`.

The number of input samples must be an integer multiple of this parameter value. Setting the period greater than `1` increases the speed of the AGC algorithm.

If you increase the period of the gain updates, you may also need to increase the step size. Similarly, if you decrease the period of the gain updates, you may also need to decrease the step size.

Step size

Specify the step size for gain updates as a double- or single-precision, real, positive scalar. The default is `0.1`.

If you increase the loop gain, the AGC responds to changes at the input signal level faster. However, gain pumping also increase.

If you increase the period of the gain updates, you may also need to increase the step size. Similarly, if you decrease the period of the gain updates, you may also need to decrease the step size.

Maximum gain in dB

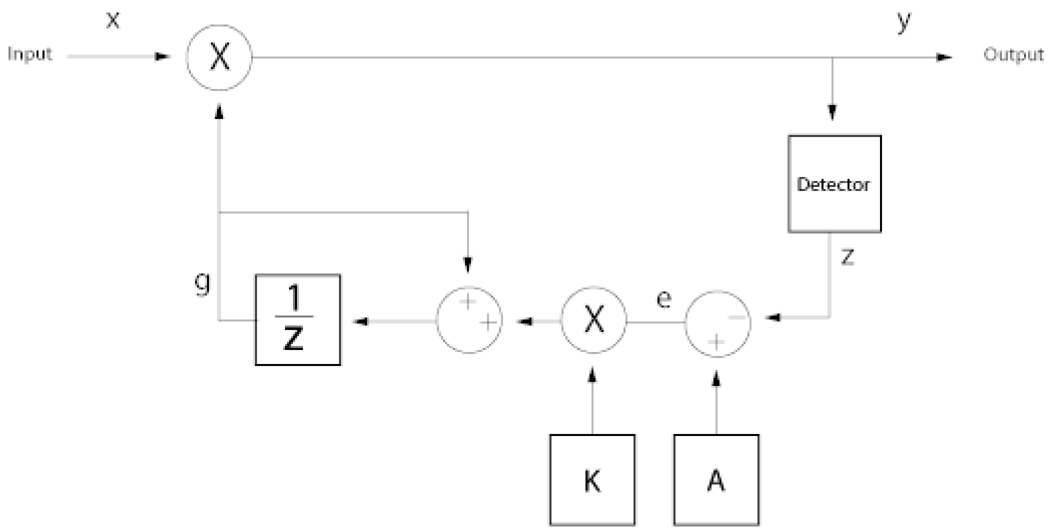
Specify the maximum gain of the AGC in decibels as a positive scalar. The default is `30`.

If the input signal to the AGC has a very low signal level, the AGC gain may increase rapidly. Use this parameter to limit the gain that the AGC applies to the input signal.

Algorithms

Linear Loop AGC

In a linear loop AGC, the detector uses its output directly to generate an error signal. After applying a step size, the AGC passes the error signal to an integrator. The output of the integrator is used as the variable gain. Linear loop AGCs are limited by their decay, or slew, characteristics. In other words, they respond to input signal increases much more quickly than they respond to input signal decreases.



$$y(n) = g(n) \cdot x(n);$$
$$e(n) = A - z(m);$$
$$g(n+1) = g(n) + K \cdot e(n);$$

where

A represents the reference value, which is 1

K represents the step size

e represents the error signal

g represents the gain

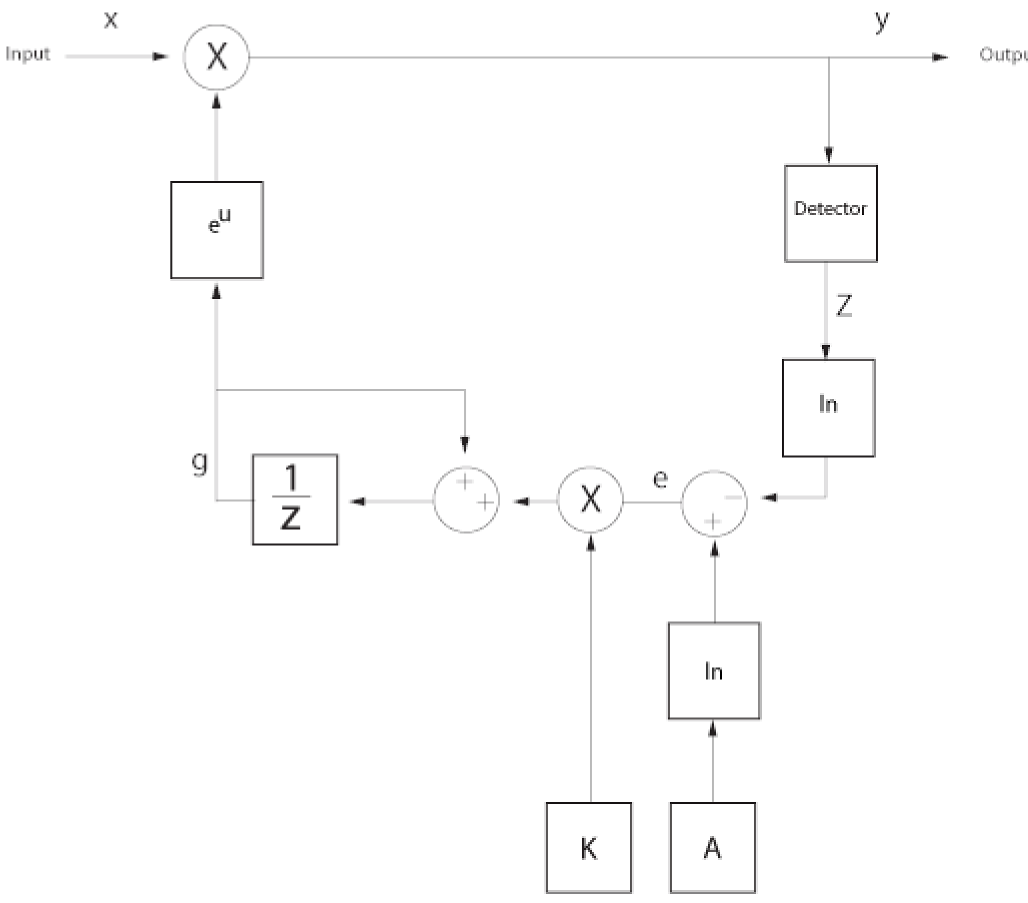
x represents the input signal

y represents the output signal

z represents the detector output

Logarithmic Loop AGC

In a logarithmic loop AGC, the logarithm of the ratio of the detector output and the reference signal represents the error signal. A logarithmic loop uses the exponential of the integrator output as the gain signal. Logarithmic loop AGCs have the same response time to both increases or decreases to the input signal amplitude.



The logarithmic loop has longer attack and decay times. However, the gain pumping of the logarithmic loop is better than that of the linear loop.

$$y(n) = e^{g(n)} \cdot x(n);$$
$$e(n) = ln(A) - \ln(z(m));$$
$$g(n+1) = g(n) + K \cdot e(n);$$

where

A represents the reference value, which is 1

K represents the step size

e represents the error signal

g represents the gain

x represents the input signal

y represents the output signal

z represents the detector output

AGC Detector

Two AGC detectors are available:

Rectifier

z = |y| when the detector represents a rectifier

$$z(m) = \frac{1}{N}\sum_{n=mN}^{(m+1)N-1} \left|y(n)\right|$$

where N represents the period of the gain updates

Square Law

z = $|y|^2$ represents the square law detector

$$z(m) = \frac{1}{N}\sum_{n=mN}^{(m+1)N-1} \left|y(n)\right|^2$$

where N represents the period of the gain updates

Performance Considerations

There are three performance criteria for AGCs:

- Attack time: The duration it takes the AGC to respond to an increase in the input amplitude.
- Decay time: The duration it takes the AGC to respond to a decrease in the input amplitude.
- Gain pumping: The variation in the gain value during steady-state operation.

Increasing the step size decreases the attack time and decay times, but it also increases gain pumping.

Examples

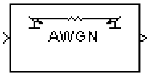
- To open an example that adaptively adjusts the received signal amplitude to approximately 1 volt, type `doc_agc_received_signal_amplitude` at the MATLAB command line.
- To open an example that compare the performance of an AGC with a rectifier detector and a square law detector, type `doc_agc_compare_rectifier_and_square_law` at the MATLAB command line.
- Top open an example that plots the effect of step size on AGC performance, type `doc_agc_plot_step_size` at the MATLAB command line.
- To open an example that plots the effect of maximum gain on burst signals, type `doc_agc_plot_max_gain` at the MATLAB command line.

AWGN Channel

Purpose Add white Gaussian noise to input signal

Library Channels

Description



The AWGN Channel block adds white Gaussian noise to a real or complex input signal. When the input signal is real, this block adds real Gaussian noise and produces a real output signal. When the input signal is complex, this block adds complex Gaussian noise and produces a complex output signal. This block inherits its sample time from the input signal.

This block accepts a scalar-valued, vector, or matrix input signal with a data type of type `single` or `double`. The output signal inherits port data types from the signals that drive the block.

Note All values of power assume a nominal impedance of 1 ohm.

Signal Processing and Input Dimensions

This block can process multichannel signals. When you set the **Input Processing** parameter to `Columns as channels (frame based)`, the block accepts an *M*-by-*N* input signal. *M* specifies the number of samples per channel and *N* specifies the number of channels. Both *M* and *N* can be equal to `1`. The block adds frames of length-*M* Gaussian noise to each of the *N* channels, using a distinct random distribution per channel.

Specifying the Variance Directly or Indirectly

You can specify the variance of the noise generated by the AWGN Channel block using one of these modes:

- `Signal to noise ratio (Eb/No)`, where the block calculates the variance from these quantities that you specify in the dialog box:
  - **Eb/No**, the ratio of bit energy to noise power spectral density
  - **Number of bits per symbol**

- **Input signal power**, the actual power of the symbols at the input of the block
- **Symbol period**

- `Signal to noise ratio (Es/No)`, where the block calculates the variance from these quantities that you specify in the dialog box:
  - **Es/No**, the ratio of signal energy to noise power spectral density
  - **Input signal power**, the actual power of the symbols at the input of the block
  - **Symbol period**
- `Signal to noise ratio (SNR)`, where the block calculates the variance from these quantities that you specify in the dialog box:
  - **SNR**, the ratio of signal power to noise power
  - **Input signal power**, the actual power of the samples at the input of the block
- `Variance from mask`, where you specify the variance in the dialog box. The value must be positive.
- `Variance from port`, where you provide the variance as an input to the block. The variance input must be positive, and its sampling rate must equal that of the input signal.

Changing the symbol period in the `AWGN Channel` block affects the variance of the noise added per sample, which also causes a change in the final error rate.

$$NoiseVariance = \frac{SignalPower \times SymbolPeriod}{SampleTime \times 10^{\frac{Es/No}{10}}}$$

A good rule of thumb for selecting the **Symbol period** value is to set it to be what you model as the symbol period in the model. The value would depend upon what constitutes a symbol and what the oversampling applied to it is (e.g., a symbol could have 3 bits and be oversampled by 4).

In both `Variance from mask` mode and `Variance from port` mode, these rules describe how the block interprets the variance:

- If the variance is a scalar, then all signal channels are uncorrelated but share the same variance.
- If the variance is a vector whose length is the number of channels in the input signal, then each element represents the variance of the corresponding signal channel.

Note If you apply complex input signals to the AWGN Channel block, then it adds complex zero-mean Gaussian noise with the calculated or specified variance. The variance of each of the quadrature components of the complex noise is half of the calculated or specified value.

Relationship Among Eb/No, Es/No, and SNR Modes

For complex input signals, the AWGN Channel block relates E_b/N_0, E_s/N_0, and SNR according to the following equations:

$$E_s/N_0 = (T_{sym}/T_{samp}) \cdot \mathrm{SNR}$$

$$E_s/N_0 = E_b/N_0 + 10\log_{10}(k) \quad \text{in dB}$$

where

- E_s = Signal energy (Joules)
- E_b = Bit energy (Joules)
- N_0 = Noise power spectral density (Watts/Hz)
- T_{sym} is the **Symbol period** parameter of the block in `Es/No` mode
- k is the number of information bits per input symbol

- T_{samp} is the inherited sample time of the block, in seconds

For real signal inputs, the AWGN Channel block relates E_s/N_0 and SNR according to the following equation:

$$E_s/N_0 = 0.5\ (T_{sym}/T_{samp}) \cdot \text{SNR}$$

Note that the equation for the real case differs from the corresponding equation for the complex case by a factor of 2. This is so because the block uses a noise power spectral density of $N_0/2$ Watts/Hz for real input signals, versus N_0 Watts/Hz for complex signals.

For more information about these quantities, see "AWGN Channel Noise Level" in the Communications System Toolbox documentation.

Tunable Block Parameters

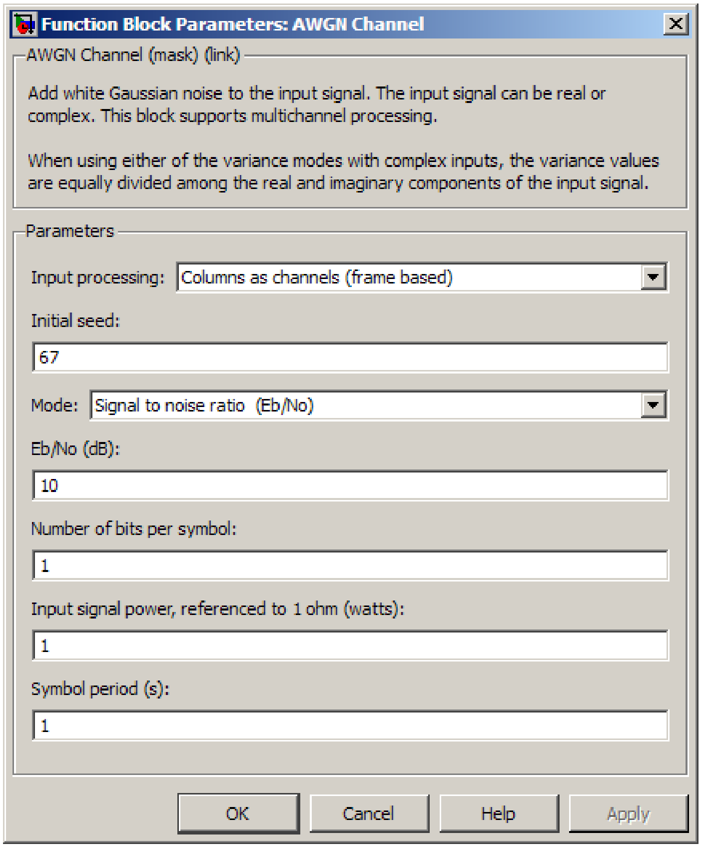
The following table indicates which parameters are tunable, for different block modes.

Mode	Tunable Parameters
`Eb/No`	**Eb/No**, **Input signal power**
`Es/No`	**Es/No**, **Input signal power**
`SNR`	**SNR**, **Input signal power**
`Variance from mask`	**Variance**

You can tune parameters in normal mode, Accelerator mode and the Rapid Accelerator mode.

If you use the Simulink Coder™ rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameters listed in the previous table without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

Dialog Box



Input processing

Specify how the block processes the input signal. 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.

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 *Communications System Toolbox Release Notes* for more information.

Initial seed
The seed for the Gaussian noise generator.

This block uses the DSP System Toolbox Random Source block to generate noise. Random numbers are generated using the Ziggurat method. The **Initial seed** parameter in this block initializes the noise generator. **Initial seed** can be either a scalar or a vector with a length that matches the number of channels in the input signal. Each time you run a simulation, this block outputs the same signal. The first time you run the simulation, the block randomly selects an initial seed. The block reuses the same initial seeds every time you rerun the simulation.

This property is a tunable and allows you to specify different seed values for each DLL build.

Mode
The mode by which you specify the noise variance: `Signal to noise ratio (Eb/No)`, `Signal to noise ratio (Es/No)`, `Signal to noise ratio (SNR)`, `Variance from mask`, or `Variance from port`.

Eb/No (dB)
The ratio of information (i.e., without channel coding) bit energy per symbol to noise power spectral density, in decibels. This field appears only if **Mode** is set to `Eb/No`.

Es/No (dB)
The ratio of information (i.e., without channel coding) symbol energy per symbol to noise power spectral density, in decibels. This field appears only if **Mode** is set to `Es/No`.

SNR (dB)
The ratio of signal power to noise power, in decibels. This field appears only if **Mode** is set to `SNR`.

Number of bits per symbol
The number of bits in each input symbol. This field appears only if **Mode** is set to `Eb/No`.

Input signal power, referenced to 1 ohm (watts)
The mean square power of the input symbols (if **Mode** is `Eb/No` or `Es/No`) or input samples (if **Mode** is `SNR`), in watts. This field appears only if **Mode** is set to `Eb/No`, `Es/No`, or `SNR`.

Symbol period (s)
The duration of an information channel (i.e., without channel coding) symbol, in seconds. This field appears only if **Mode** is set to `Eb/No` or `Es/No`.

Variance
The variance of the white Gaussian noise. This field appears only if **Mode** is set to `Variance from mask`.

Examples

Many documentation examples use this block, including:

- Gray Coded 8-PSK (`EbNo` mode)
- Phase Noise Effects in 256-QAM (`SNR` mode)
- Discrete Multitone Signaling (`Variance from mask` mode)
- "Filter Using Simulink Raised Cosine Filter Blocks"

See Also

Random Source (DSP System Toolbox documentation)

Reference

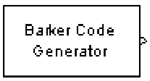
[1] Proakis, John G., *Digital Communications*, 4th Ed., McGraw-Hill, 2001.

Barker Code Generator

Purpose Generate Barker Code

Library Sequence Generators sublibrary of Comm Sources

Description



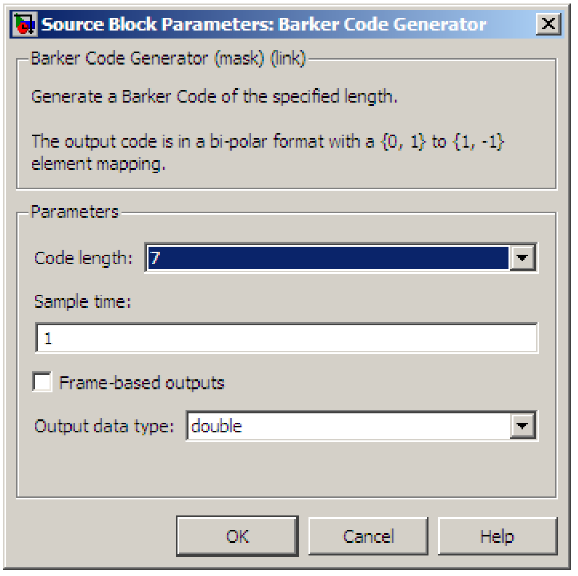
Barker codes, which are subsets of PN sequences, are commonly used for frame synchronization in digital communication systems. Barker codes have length at most 13 and have low correlation sidelobes. A correlation sidelobe is the correlation of a codeword with a time-shifted version of itself. The correlation sidelobe, C_k, for a k-symbol shift of an N-bit code sequence, $\{X_j\}$, is given by

$$C_k = \sum_{j=1}^{N-k} X_j X_{j+k}$$

where X_j is an individual code symbol taking values +1 or -1 for j=1, 2, 3,..., N, and the adjacent symbols are assumed to be zero.

The Barker Code Generator block provides the codes listed in the following table:

Code length	Barker Code
1	[-1]
2	[-1 1];
3	[-1 -1 1]
4	[-1 -1 1 -1]
5	[-1 -1 -1 1 -1]
7	[-1 -1 -1 1 1 -1 1]
11	[-1 -1 -1 1 1 1 -1 1 1 -1 1]
13	[-1 -1 -1 -1 -1 1 1 -1 -1 1 -1 1 -1]



Dialog Box

Code length
: The length of the Barker code.

Sample time
: Period of each element of the output signal.

Frame-based outputs
: Determines whether the block's output is frame-based or sample-based.

Samples per frame
: The number of samples in a frame-based output signal. This field appears if you select **Frame-based outputs**.

Output data type
: The output type of the block can be specified as an `int8` or `double`. By default, the block sets this to `double`.

See Also

PN Sequence Generator

Purpose

Implement baseband phase-locked loop

Library

Components sublibrary of Synchronization

Description

The Baseband PLL (phase-locked loop) block is a feedback control system that automatically adjusts the phase of a locally generated signal to match the phase of an input signal. Unlike the Phase-Locked Loop block, this block uses a baseband method and does not depend on a carrier frequency.

This PLL has these three components:

- An integrator used as a phase detector.
- A filter. You specify the filter's transfer function using the **Lowpass filter numerator** and **Lowpass filter denominator** parameters. Each is a vector that gives the respective polynomial's coefficients in order of descending powers of *s*.

  To design a filter, you can use the Signal Processing Toolbox™ functions `cheby1`, and `cheby2`. The default filter is a Chebyshev type II filter whose transfer function arises from the command below.

  ```
  [num, den] = cheby2(3,40,100,'s')
  ```

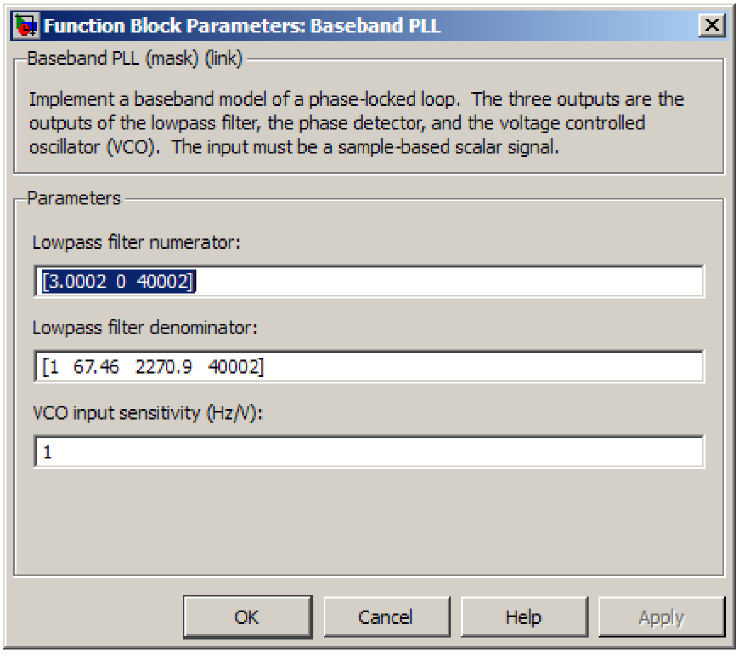
- A voltage-controlled oscillator (VCO). You specify the sensitivity of the VCO signal to its input using the **VCO input sensitivity** parameter. This parameter, measured in Hertz per volt, is a scale factor that determines how much the VCO shifts from its quiescent frequency.

This block accepts a sample-based scalar signal. The input signal represents the received signal. The three output ports produce:

- The output of the filter
- The output of the phase detector
- The output of the VCO

This model is nonlinear; for a linearized version, use the Linearized Baseband PLL block.

Dialog Box



Lowpass filter numerator

The numerator of the lowpass filter's transfer function, represented as a vector that lists the coefficients in order of descending powers of s.

Lowpass filter denominator

The denominator of the lowpass filter's transfer function, represented as a vector that lists the coefficients in order of descending powers of s.

VCO input sensitivity (Hz/V)

This value scales the input to the VCO and, consequently, the shift from the VCO's quiescent frequency.

See Also

Linearized Baseband PLL, Phase-Locked Loop

References

For more information about phase-locked loops, see the works listed in "Selected Bibliography for Synchronization" in *Communications System Toolbox User's Guide*.

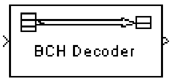
Purpose

Decode BCH code to recover binary vector data

Library

Block sublibrary of Error Detection and Correction

Description



The BCH Decoder block recovers a binary message vector from a binary BCH codeword vector. For proper decoding, the first two parameter values in this block should match the parameters in the corresponding BCH Encoder block.

This block accepts a column vector input signal with an integer multiple of (*N* - the number of punctures) elements. Each group of *N* input elements represents one codeword to be decoded. The values of (*N* + shortening length) and (*K* + shortening length) must produce a valid narrow-sense BCH code.

If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

For a given codeword length *N*, only specific message lengths *K* are valid for a BCH code. For a full length BCH code, *N* must be of the form 2^M-1, where $3 \le M \le 16$. If *N* is less than 2^M-1, the block assumes that the code has been shortened by length 2^{M-1} - *N*. However, if *N* is greater than or equal to 2^{M-1}, **Primitive polynomial** must be specified to appropriately set the value of *M*.

No known analytic formula describes the relationship among the codeword length, message length, and error-correction capability. For a list of some valid values of *K* corresponding to values of *N* up to 511, see the BCH Encoder reference page in the Communications System Toolbox documentation.

The primitive and generator polynomials may be specified in their respective fields, which appear after selecting their corresponding check boxes.

To have the block output error information, select **Output number of corrected errors**. Selecting this option causes a second output port to appear. The second output is the number of errors detected during decoding of the codeword. A negative integer indicates that the block detected more errors than it could correct using the coding scheme.

In the case of a decoder failure, the message portion of the decoder input is returned unchanged as the decoder output.

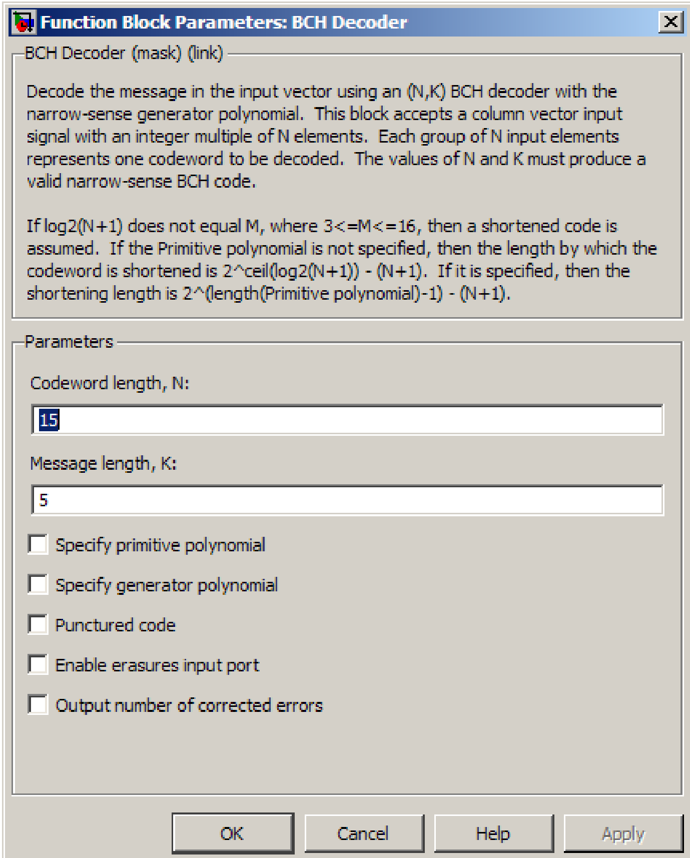
The sample times of all input and output signals are equal.

For information about the data types each block port supports, see the “Supported Data Type” on page 2-44 table on this page.

Punctured Codes

This block supports puncturing when you select **Punctured code**. This selection enables the **Puncture vector** parameter, which takes in a binary vector to specify the puncturing pattern. For a puncture vector, `1` represents that the data symbol passes unaltered, and `0` represents that the data symbol gets punctured, or removed, from the data stream. This convention applies for both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Note `1`s and `0`s have precisely opposite meanings for the puncture and erasure vectors. For an erasure vector, `1` means that the data symbol is to be replaced with an erasure symbol, and `0` means that the data symbol is passed unaltered. This convention is carried for both the encoder and the decoder.



Dialog Box

Codeword length, N
: The codeword length.

Message length, K
: The message length.

Specify primitive polynomial
: Selecting this check box enables the **Primitive polynomial** field.

Primitive polynomial

A row vector that represents the binary coefficients of the primitive polynomial in order of descending powers.

This field defaults to `de2bi(primpoly(4, 'nodisplay'), 'left-msb')`, corresponding to a (15,5) code.

This parameter appears only when you select **Specify primitive polynomial**.

Specify generator polynomial

Selecting this check box enables the **Generator polynomial** field.

Generator polynomial

A row vector that represents the binary coefficients of the generator polynomial in order of descending powers.

The length of the **Generator polynomial** must be N-K+1.

This field defaults to `bchgenpoly(15,5)`.

This parameter appears only when you select **Specify generator polynomial**.

Disable generator polynomial checking

Each time a model initializes, the block performs a polynomial check. This check verifies that $X^N + 1$ is divisible by the user-defined generator polynomial, where N represents the full code word length. Selecting this check box disables the polynomial check. For larger codes, disabling the check speeds up the simulation process. You should always run the check at least once before disabling this feature.

This check box appears only when you select **Specify generator polynomial**.

Puncture code

Selecting this check box enables the field **Puncture vector**.

Puncture vector

This parameter appears only when you select **Puncture code**.

A column vector of length *N-K*. In the **Puncture vector**, a value of `1` represents that the data symbol passes unaltered, and `0` represents that the data symbol gets punctured, or removed, from the data stream.

The default value is `[ones(8,1); zeros(2,1)]`.

Enable erasures input port

Selecting this check box will open the `Era` port.

Through the `Era` port, you can input a binary column vector the same size as the codeword input.

Erasure values of `1` correspond to erased bits in the same position in the codeword, and values of `0` correspond to bits that are not erased.

Output number of corrected errors

Selecting this check box gives the block an additional output port, Err, which indicates the number of errors the block corrected in the input codeword.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Era	• Double-precision floating point • Boolean
Err	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

BCH Encoder

References

[1] Wicker, Stephen B., *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, N.J., Prentice Hall, 1995.

[2] Berlekamp, Elwyn R., *Algebraic Coding Theory*, New York, McGraw-Hill, 1968.

[3] Clark, George C., Jr., and J. Bibb Cain, *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

See Also

`bchdec` (in Communications System Toolbox documentation)

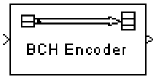
Purpose

Create BCH code from binary vector data

Library

Block sublibrary of Error Detection and Correction

Description



The BCH Encoder block creates a BCH code with message length K and codeword length (N - number of punctures). You specify both N and K directly in the dialog box.

This block accepts a column vector input signal with an integer multiple of K elements. Each group of K input elements represents one message word to be encoded.

If the encoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

For a given codeword length N, only specific message lengths K are valid for a BCH code. For a full length BCH code, N must be of the form 2^M-1, where $3 \le M \le 16$. If N is less than 2^M-1, the block assumes that the code has been shortened by length $2^{M-1} - N$. However, if N is greater than or equal to 2^{M-1}, **Primitive polynomial** must be specified to appropriately set the value of M.

No known analytic formula describes the relationship among the codeword length, message length, and error-correction capability. The tables below list valid [n,k] pairs for small values of n, as well as the corresponding values of the error-correction capability, t.

n	k	t
7	4	1

n	k	t
15	11	1
15	7	2
15	5	3

n	k	t
31	26	1
31	21	2
31	16	3
31	11	5
31	6	7

n	k	t
63	57	1
63	51	2
63	45	3
63	39	4
63	36	5
63	30	6
63	24	7
63	18	10
63	16	11
63	10	13
63	7	15

n	k	t
127	120	1
127	113	2
127	106	3
127	99	4
127	92	5
127	85	6
127	78	7
127	71	9
127	64	10
127	57	11
127	50	13
127	43	14
127	36	15
127	29	21
127	22	23
127	15	27
127	8	31

n	k	t
255	247	1
255	239	2
255	231	3
255	223	4

n	k	t
255	215	5
255	207	6
255	199	7
255	191	8
255	187	9
255	179	10
255	171	11
255	163	12
255	155	13
255	147	14
255	139	15
255	131	18
255	123	19
255	115	21
255	107	22
255	99	23
255	91	25
255	87	26
255	79	27
255	71	29
255	63	30
255	55	31
255	47	42
255	45	43
255	37	45

n	k	t
255	29	47
255	21	55
255	13	59
255	9	63

n	k	t
511	502	1
511	493	2
511	484	3
511	475	4
511	466	5
511	457	6
511	448	7
511	439	8
511	430	9
511	421	10
511	412	11
511	403	12
511	394	13
511	385	14
511	376	15
511	367	16
511	358	18

n	k	t
511	349	19
511	340	20
511	331	21
511	322	22
511	313	23
511	304	25
511	295	26
511	286	27
511	277	28
511	268	29
511	259	30
511	250	31
511	241	36
511	238	37
511	229	38
511	220	39
511	211	41
511	202	42
511	193	43
511	184	45
511	175	46
511	166	47
511	157	51
511	148	53
511	139	54

n	k	t
511	130	55
511	121	58
511	112	59
511	103	61
511	94	62
511	85	63
511	76	85
511	67	87
511	58	91
511	49	93
511	40	95
511	31	109
511	28	111
511	19	119
511	10	121

The primitive and generator polynomials may be specified in their respective fields, which appear after selecting their corresponding check boxes.

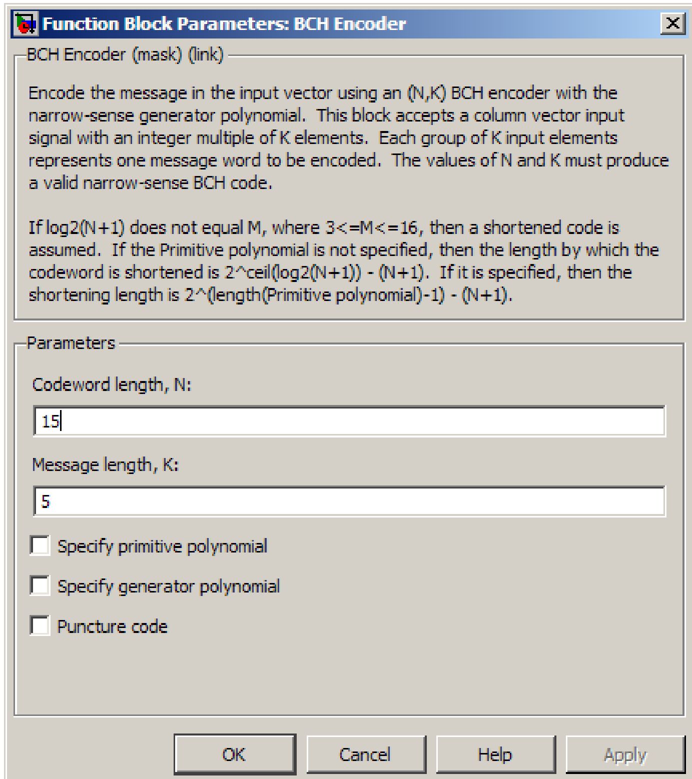
For information about the data types each block port supports, see the "Supported Data Type" on page 2-57 table on this page.

Puncture Codes

This block supports puncturing when you select the **Puncture code** parameter. This selection enables the **Puncture vector** parameter, which takes in a binary vector to specify the puncturing pattern. For a puncture vector, 1 represents that the data symbol passes unaltered, and 0 represents that the data symbol gets punctured, or removed, from the data stream. This convention is carried for both the encoder and the decoder. For more information, see "Shortening, Puncturing, and Erasures".

Note 1s and 0s have precisely opposite meanings for the puncture and erasure vectors. For an erasure vector, 1 means that the data symbol is to be replaced with an erasure symbol, and 0 means that the data symbol is passed unaltered. This convention is carried for both the encoder and the decoder.

Dialog Box



Codeword length, N
: The codeword length.

Message length, K
: The message length.

Specify primitive polynomial
: Selecting this check box enables the **Primitive polynomial** field.

Primitive polynomial
: A row vector that represents the binary coefficients of the primitive polynomial in order of descending powers.

This field defaults to `de2bi(primpoly(4, 'nodisplay'), 'left-msb')`, corresponding to a (15,5) code.

This parameter applies only when you select **Specify primitive polynomial**.

Specify generator polynomial
: Selecting this check box enables the **Generator polynomial** field.

Generator polynomial
: A row vector that represents the binary coefficients of the generator polynomial in order of descending powers.

  The length of the **Generator polynomial** must be N-K+1.

  This field defaults to `bchgenpoly(15,5)`.

  This parameter applies only when you select **Specify generator polynomial**.

Disable generator polynomial checking
: This check box appears only when you select **Specify generator polynomial**.

  Each time a model initializes, the block performs a polynomial check. This check verifies that $X^N + 1$ is divisible by the user-defined generator polynomial, where N represents the full code word length. Selecting this check box disables the polynomial check. For larger codes, disabling the check speeds up the simulation process. You should always run the check at least once before disabling this feature.

Puncture code
: Selecting this check box enables the **Puncture vector** field.

Puncture vector
: A column vector of length N-K. In the **Puncture vector**, a value of `1` represents that the data symbol passes unaltered, and `0`

represents that the data symbol gets punctured, or removed, from the data stream.

The field defaults to `[ones(8,1); zeros(2,1)]`.

This parameter applies only when you select **Puncture code**.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

BCH Decoder

References

[1] Clark, George C., Jr., and J. Bibb Cain, *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

See Also

`bchenc` (in Communications System Toolbox documentation)

Bernoulli Binary Generator

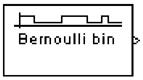
Purpose

Generate Bernoulli-distributed random binary numbers

Library

Random Data Sources sublibrary of Comm Sources

Description



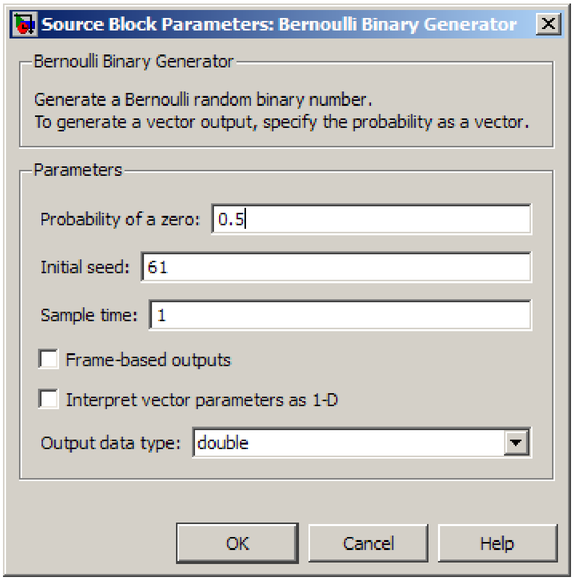
The Bernoulli Binary Generator block generates random binary numbers using a Bernoulli distribution. The Bernoulli distribution with parameter p produces zero with probability p and one with probability 1-p. The Bernoulli distribution has mean value 1-p and variance p(1-p). The **Probability of a zero** parameter specifies p, and can be any real number between zero and one.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples per frame**, and **Interpret vector parameters as 1-D** parameters. See "Sources and Sinks" in *Communications System Toolbox User's Guide* for more details.

The number of elements in the **Initial seed** and **Probability of a zero** parameters becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. Also, the shape (row or column) of the **Initial seed** and **Probability of a zero** parameters becomes the shape of a sample-based two-dimensional output signal.

Dialog Box



Probability of a zero

The probability with which a zero output occurs.

Initial seed

The initial seed value for the random number generator. The seed can be either a vector of the same length as the **Probability of a zero** parameter, or a scalar.

Sample time

The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs

Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame
: The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D
: If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type
: The output type of the block can be specified as a `boolean`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, or `double`. By default, the block sets this to `double`. Single outputs may lead to different results when compared with double outputs for the same set of parameters.

See Also

Random Integer Generator, Binary Symmetric Channel; `randint` (in Communications System Toolboxdocumentation), `rand` (built-in MATLAB function)

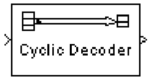
Purpose

Decode systematic cyclic code to recover binary vector data

Library

Block sublibrary of Error Detection and Correction

Description



The Binary Cyclic Decoder block recovers a message vector from a codeword vector of a binary systematic cyclic code. For proper decoding, the parameter values in this block should match those in the corresponding Binary Cyclic Encoder block.

If the cyclic code has message length K and codeword length N, then N must have the form 2^M-1 for some integer M greater than or equal to 3.

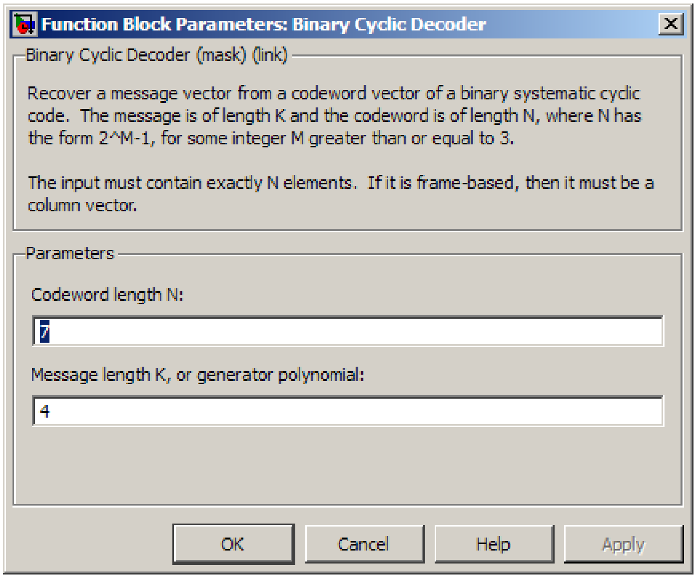
This block accepts a column vector input signal containing N elements. The output signal is a column vector containing K elements.

You can determine the systematic cyclic coding scheme in one of two ways:

- To create an [N,K] code, enter N and K as the first and second dialog parameters, respectively. The block computes an appropriate generator polynomial, namely, `cyclpoly(N,K,'min')`.
- To create a code with codeword length N and a particular degree-(N-K) binary *generator polynomial*, enter N as the first parameter and a binary vector as the second parameter. The vector represents the generator polynomial by listing its coefficients in order of ascending exponents. You can create cyclic generator polynomials using the Communications System Toolbox `cyclpoly` function.

For information about the data types each block port supports, see the “Supported Data Type” on page 2-63 table on this page.

Dialog Box



Codeword length N
: The codeword length N, which is also the input vector length.

Message length K, or generator polynomial
: Either the message length, which is also the output vector length; or a binary vector that represents the generator polynomial for the code.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point

Pair Block

Binary Cyclic Encoder

See Also

`cyclpoly` (Communications Toolbox)

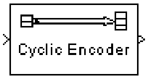
Purpose

Create systematic cyclic code from binary vector data

Library

Block sublibrary of Error Detection and Correction

Description



The Binary Cyclic Encoder block creates a systematic cyclic code with message length *K* and codeword length *N*. The number *N* must have the form 2^M-1, where *M* is an integer greater than or equal to 3.

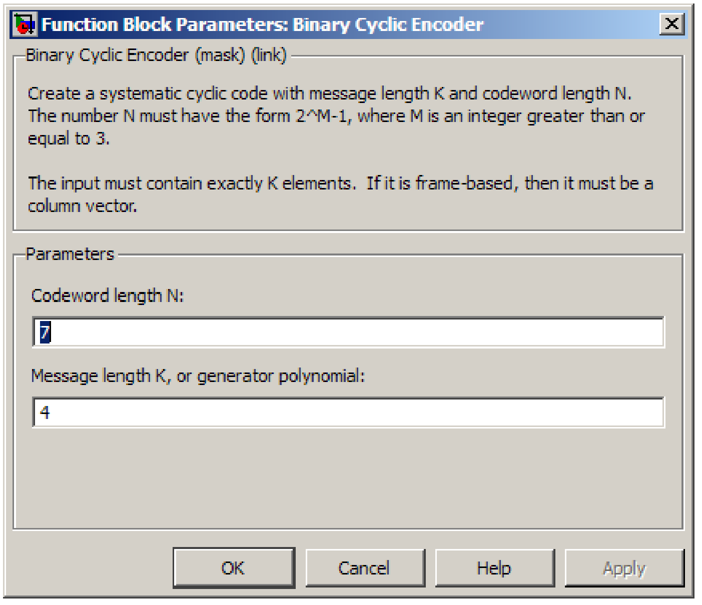
This block accepts a column vector input signal containing *K* elements. The out put signal is a column vector containing *N* elements.

You can determine the systematic cyclic coding scheme in one of two ways:

- To create an [N,K] code, enter N and K as the first and second dialog parameters, respectively. The block computes an appropriate generator polynomial, namely, `cyclpoly(N,K,'min')`.
- To create a code with codeword length *N* and a particular degree-(N-K) binary *generator polynomial*, enter *N* as the first parameter and a binary vector as the second parameter. The vector represents the generator polynomial by listing its coefficients in order of ascending exponents. You can create cyclic generator polynomials using the Communications System Toolbox `cyclpoly` function.

For information about the data types each block port supports, see the "Supported Data Type" on page 2-66 table on this page.

Dialog Box



Codeword length N

The codeword length, which is also the output vector length.

Message length K, or generator polynomial

Either the message length, which is also the input vector length; or a binary vector that represents the generator polynomial for the code.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point

Pair Block

Binary Cyclic Decoder

See Also

`cyclpoly` (in the Communications System Toolbox documentation)

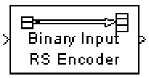
Purpose

Create Reed-Solomon code from binary vector data

Library

Block sublibrary of Error Detection and Correction

Description



The Binary-Input RS Encoder block creates a Reed-Solomon code with message length, K, and codeword length, (N - number of punctures). You specify both N and K directly in the dialog box. The symbols for the code are binary sequences of length M, corresponding to elements of the Galois field $GF(2^M)$, where the first bit in each sequence is the most significant bit. Restrictions on M and N are given in "Restrictions on the M and the Codeword Length N" on page 2-68 below. The difference N-K must be an even integer.

This block can output shortened codewords when N and K are appropriately specified. To specify output codewords that are shortened by a length S, N and K must be specified in the dialog box as $N_{full} - S$ and $K_{full} - S$, where N_{full} and K_{full} are the N and K of an unshortened code. If $S < (N_{full} + 1)/2$, the encoder can automatically determine the value of N_{full} and K_{full}. However, if $S \geq (N_{full} + 1)/2$, **Primitive polynomial** must be specified in order to properly define the extension field for the code.

The input and output are binary-valued signals that represent messages and codewords, respectively. This block accepts a column vector input signal with a length that is an integer multiple of M*K. This block outputs a column vector with a length that is the same integer multiple of M*(N - number of punctures). The block inherits the output data type from the input. For information about the data types each block port supports, see the "Supported Data Type" on page 2-73 table on this page.

For more information on representing data for Reed-Solomon codes, see the section "Integer Format (Reed-Solomon Only)" in *Communications System Toolbox User's Guide*.

If the encoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

The default value of M is the smallest integer that is greater than or equal to log2(N+1), that is, `ceil(log2(N+1))`. You can change the value of M from the default by specifying the primitive polynomial for GF(2^M), as described in "Specifying the Primitive Polynomial" on page 2-68 below. If N is less than 2^M-1, the block assumes that the code has been shortened by length 2^{M-1} - N.

Each M*K input bits represent K integers between 0 and 2^M-1. Similarly, each M*(N - number of punctures) output bits represent N integers between 0 and 2^M-1. These integers in turn represent elements of the Galois field GF(2^M).

An (N,K) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (*not* bit errors) in each codeword.

Specifying the Primitive Polynomial

You can specify the primitive polynomial that defines the finite field GF(2^M), corresponding to the integers that form messages and codewords. To do so, first select **Specify primitive polynomial**. Then, set **Primitive polynomial** to a binary row vector that represents a primitive polynomial over GF(2) of degree M, in descending order of powers. For example, to specify the polynomial x^3+x+1, enter the vector `[1 0 1 1]`.

If you do not select **Specify primitive polynomial**, the block uses the default primitive polynomial of degree M = ceil(log2(N+1)). You can display the default polynomial by entering `primpoly(ceil(log2(N+1)))` at the MATLAB prompt.

Restrictions on the M and the Codeword Length N

The restrictions on the degree M of the primitive polynomial and the codeword length N are as follows:

- If you do not select **Specify primitive polynomial**, N must lie in the range $3 < N \le 2^{16}-1$.
- If you do select **Specify primitive polynomial**, N must lie in the range $3 \le N \le 2^{16}-1$ and M must lie in the range $3 \le M \le 16$.

Specifying the Generator Polynomial

You can specify the generator polynomial for the Reed-Solomon code. To do so, first select **Specify generator polynomial**. Then, in the **Generator polynomial** field, enter an integer row vector whose entries are between 0 and 2^M-1. The vector represents a polynomial, in descending order of powers, whose coefficients are elements of $GF(2^M)$ represented in integer format. See the section "Integer Format (Reed-Solomon Only)" for more information about integer format. The generator polynomial must be equal to a polynomial with a factored form

$$g(x) = (x+A^b)(x+A^{b+1})(x+A^{b+2})...(x+A^{b+N-K-1})$$

where A is the primitive element of the Galois field over which the input message is defined, and *b* is a non-negative integer.

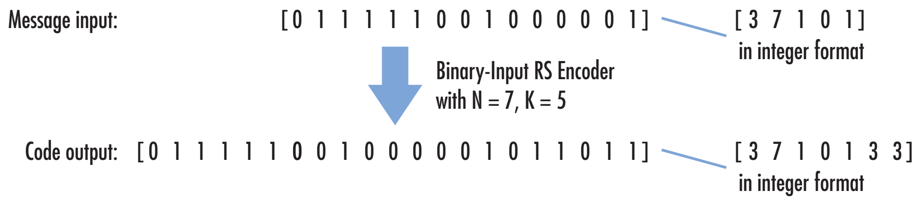
If you do not select **Specify generator polynomial**, the block uses the default generator polynomial, corresponding to *b*=1, for Reed-Solomon encoding. You can display the default generator polynomial by entering `rsgenpoly(N1,K1)`, where `N1=2^M-1` and `K1=K+(N1-N)`, at the MATLAB prompt, if you are using the default primitive polynomial. If the **Specify primitive polynomial** box is selected, and you specify the primitive polynomial specified as `poly`, the default generator polynomial is `rsgenpoly(N1,K1,poly)`.

Puncture Codes

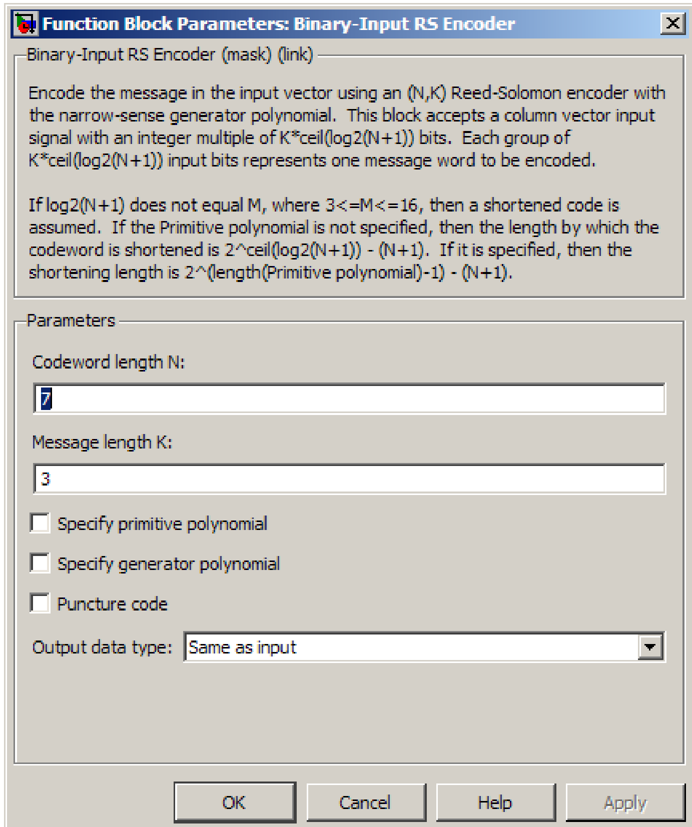
The block supports puncturing when you select the **Puncture code** parameter. This enables the **Puncture vector** parameter, which takes in a binary vector to specify the puncturing pattern. For a puncture vector, `1` represents that the data symbol passes unaltered, and `0` represents that the data symbol gets punctured, or removed, from the data stream. This convention is carried for both the encoder and the decoder. For more information, see "Shortening, Puncturing, and Erasures".

Examples

Suppose M = 3, N = 2^3-1 = 7, and K = 5. Then a message is a binary vector of length 15 that represents 5 three-bit integers. A corresponding codeword is a binary vector of length 21 that represents 7 three-bit integers. The following figure shows the codeword that would result from a particular message word. The integer format equivalents illustrate that the highest order bit is at the left.



Dialog Box



Codeword length N

The codeword length. The output has vector length NC*M*(N - NP), where NC is the number of codewords being output, and NP is the number of punctures per codeword.

Message length K

The message length. The input has vector length NM*M*K, where NM is the number of messages per frame being input.

Specify primitive polynomial

Selecting this check box enables the field **Primitive polynomial**.

Primitive polynomial

This field is available only when **Specify primitive polynomial** is selected.

Binary row vector representing the primitive polynomial in descending order of powers.

Specify generator polynomial

Selecting this check box enables the field **Generator polynomial**.

Generator polynomial

This field is available only when **Specify generator polynomial** is selected.

Integer row vector, whose entries are in the range from 0 to 2^M-1, representing the generator polynomial in descending order of powers.

Puncture code

Selecting this check box enables the field **Puncture vector**.

Puncture vector

This field is available only when **Puncture code** is selected.

A column vector of length N-K. A value of `1` in the **Puncture vector** corresponds to an M-bit symbol that is not punctured, and a `0` corresponds to an M-bit symbol that is punctured.

The default value is `[ones(2,1); zeros(2,1)]`.

Output data type

The output type of the block can be specified as `Same as input`, `boolean`, or `double`. By default, the block sets this to `Same as input`.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • 1-bit unsigned integer (ufix(1))
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • 1-bit unsigned integer (ufix(1))

Pair Block

Binary-Output RS Decoder

See Also

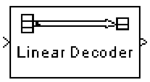
Integer-Input RS Encoder

Binary Linear Decoder

Purpose Decode linear block code to recover binary vector data

Library Block sublibrary of Error Detection and Correction

Description



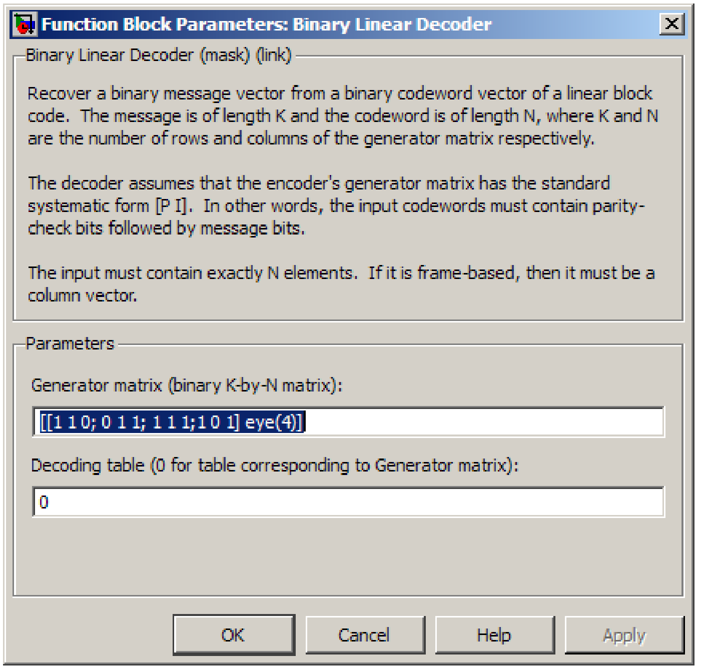
The Binary Linear Decoder block recovers a binary message vector from a binary codeword vector of a linear block code.

The **Generator matrix** parameter is the generator matrix for the block code. For proper decoding, this should match the **Generator matrix** parameter in the corresponding Binary Linear Encoder block. If N is the codeword length of the code, then **Generator matrix** must have N columns. If K is the message length of the code, then the **Generator matrix** parameter must have K rows.

This block accepts a column vector input signal containing N elements. This block outputs a column vector with a length of K elements.

The decoder tries to correct errors, using the **Decoding table** parameter. If **Decoding table** is the scalar 0, then the block defaults to the table produced by the Communications System Toolbox function `syndtable`. Otherwise, **Decoding table** must be a 2^{N-K}-by-N binary matrix. The rth row of this matrix is the correction vector for a received binary codeword whose syndrome has decimal integer value r-1. The syndrome of a received codeword is its product with the transpose of the parity-check matrix.

For information about the data types each block port supports, see the "Supported Data Type" on page 2-76 table on this page.



Dialog Box

Generator matrix

Generator matrix for the code; same as in Binary Linear Encoder block.

Decoding table

Either a 2^{N-K}-by-N matrix that lists correction vectors for each codeword's syndrome; or the scalar 0, in which case the block defaults to the table corresponding to the **Generator matrix** parameter.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point

Pair Block

Binary Linear Encoder

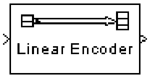
Purpose

Create linear block code from binary vector data

Library

Block sublibrary of Error Detection and Correction

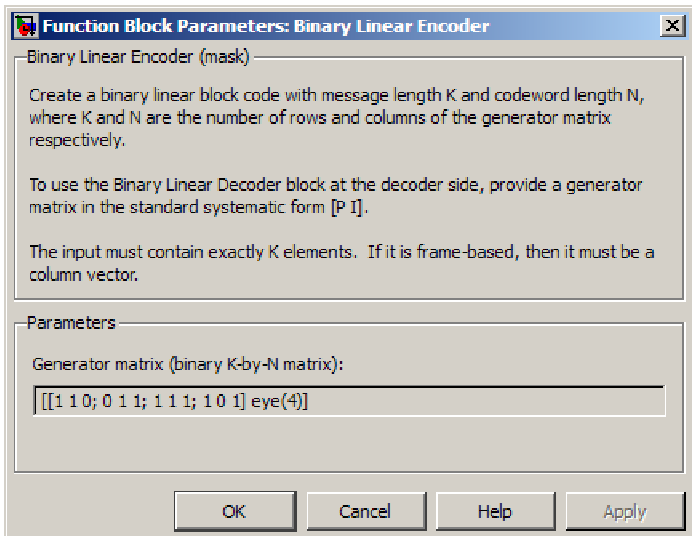
Description



The Binary Linear Encoder block creates a binary linear block code using a generator matrix that you specify. If K is the message length of the code, then the **Generator matrix** parameter must have K rows. If N is the codeword length of the code, then **Generator matrix** must have N columns.

This block accepts a column vector input signal containing K elements. This block outputs a column vector with a length of N elements. For information about the data types each block port supports, see "Supported Data Type" on page 2-78.

Dialog Box



Generator matrix

A K-by-N matrix, where K is the message length and N is the codeword length.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point

Pair Block

Binary Linear Decoder

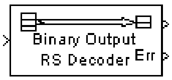
Purpose

Decode Reed-Solomon code to recover binary vector data

Library

Block sublibrary of Error Detection and Correction

Description



The Binary-Output RS Decoder block recovers a binary message vector from a binary Reed-Solomon codeword vector. For proper decoding, the parameter values in this block should match those in the corresponding Binary-Input RS Encoder block.

The Reed-Solomon code has message length, K, and codeword length, (N - number of punctures). You specify both N and K directly in the dialog box. The symbols for the code are binary sequences of length M, corresponding to elements of the Galois field GF(2^M), where the first bit in each sequence is the most significant bit. Restrictions on M and N are described in “Restrictions on the M and the Codeword Length N” on page 2-68. The difference N-K must be an even integer.

This block can output shortened codewords when N and K are appropriately specified. To specify output codewords that are shortened by a length S, N and K must be specified in the dialog box as $N_{full} - S$ and $K_{full} - S$, where N_{full} and K_{full} are the N and K of an unshortened code. If $S < (N_{full} + 1)/2$, the encoder can automatically determine the value of N_{full} and K_{full}. However, if $S \geq (N_{full} + 1)/2$, **Primitive polynomial** must be specified in order to properly define the extension field for the code.

The input and output are binary-valued signals that represent codewords and messages, respectively. This block accepts a column vector input signal with a length that is an integer multiple of M * (N - number of punctures). This block outputs a column vector with a length that is the same integer multiple of M*K. The output signal inherits its data type from the input signal. For information about the data types each block port supports, see the “Supported Data Type” on page 2-84 table on this page.

For more information on representing data for Reed-Solomon codes, see “Integer Format (Reed-Solomon Only)” in *Communications System Toolbox User’s Guide.*

If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

The default value of M is `ceil(log2(N+1))`, that is, the smallest integer greater than or equal to log2(N+1). You can change the value of M from the default by specifying the primitive polynomial for GF(2^M), as described in "Specifying the Primitive Polynomial" on page 2-68 below. If N is less than 2^M-1, the block uses a shortened Reed-Solomon code.

You can also specify the generator polynomial for the Reed-Solomon code, as described in "Specifying the Generator Polynomial" on page 2-69.

Each $M*K$ input bits represent K integers between 0 and 2^M-1. Similarly, each $M*(N$ - number of punctures) output bits represent N integers between 0 and 2^M-1. These integers in turn represent elements of the Galois field GF(2^M).

The second output is a vector of the number of errors detected during decoding of the codeword. A -1 indicates that the block detected more errors than it could correct using the coding scheme. An (N,K) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (*not* bit errors) in each codeword.

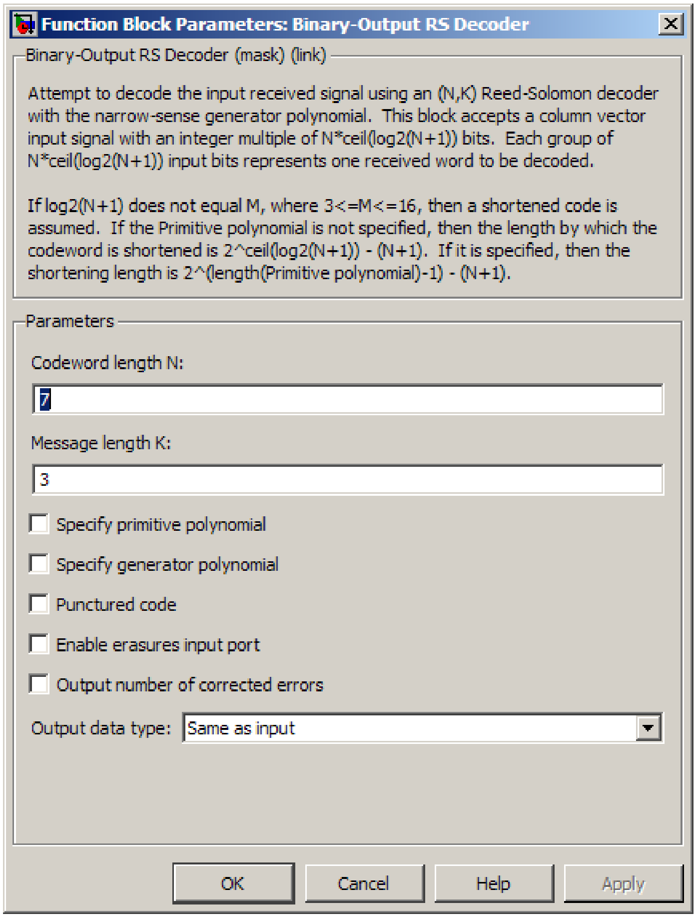
You can disable the second output by deselecting **Output port for number of corrected errors.** This removes the block's second output port.

Punctured Codes

This block supports puncturing when you select the **Punctured code** parameter. This selection enables the **Puncture vector** parameter, which takes in a binary vector to specify the puncturing pattern. For a puncture vector, `1` represents that the data symbol passes unaltered, and `0` represents that the data symbol gets punctured, or removed, from the data stream. This convention is carried for both the encoder and the decoder. For more information, see "Shortening, Puncturing, and Erasures".

Note `1`s and `0`s have precisely opposite meanings for the puncture and erasure vectors. For an erasure vector, `1` means that the data symbol is to be replaced with an erasure symbol, and `0` means that the data symbol is passed unaltered. This convention is carried for both the encoder and the decoder.

Dialog Box



Codeword length N

The codeword length. The input has vector length *NC***M**(*N* - *NP*), where *NC* is the number of codewords being output, and *NP* is the number of punctures per codeword.

Message length K

The message length. The first output has vector length *NM***M***K*, where *NM* is the number of messages per frame being output.

Specify primitive polynomial
Selecting this check box enables the **Primitive polynomial** field.

Primitive polynomial
Binary row vector representing the primitive polynomial in descending order of powers. When you provide a Primitive polynomial, the number of input bits must be an integer multiple of *K* times the order of the Primitive polynomial instead.

This parameter applies only when you select **Specify primitive polynomial**.

Specify generator polynomial
Selecting this check box enables the **Generator polynomial** field.

Generator polynomial
Integer row vector, whose entries are in the range from 0 to 2^M-1, representing the generator polynomial in descending order of powers. Each coefficient is an element of the Galois field defined by the primitive polynomial.

This parameter applies only when you select **Specify generator polynomial**.

Puncture code
Selecting this check box enables the **Puncture vector** field.

Puncture vector
A column vector of length *N*-*K*. For a puncture vector, `1` represents an *M*-bit symbol that passes unaltered, and `0` represents an *M*-bit symbol that gets punctured, or removed, from the data stream.

The default value is `[ones(2,1); zeros(2,1)]`.

This parameter applies only when you select **Punctured code**.

Enable erasures input port
Select this check to open the erasures port, `Era`.

Through the port, you can input a binary column vector that is 1/M times as long as the codeword input.

Erasure values of `1` correspond to erased symbols in the same position in the bit-packed codeword, and values of `0` correspond to nonerased symbols.

Output number of corrected errors
When you select this box, the block outputs the number of corrected errors in each word through a second output port. A decoding failure occurs when a certain received word in the input contains more than (*N*-*K*)/2 symbol errors. The value -1 indicates the corresponding position in the second output vector.

Output data type
The output type of the block can be specified as `Same as input`, `boolean`, or `double`. By default, the block sets this to `Same as input`.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • 1-bit unsigned integer (ufix(1))
Out	• Double-precision floating point • Single-precision floating point • Boolean

Port	Supported Data Types
	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • 1-bit unsigned integer (ufix(1))
Era	• Double-precision floating point • Boolean
Err	• Double-precision floating point

Algorithm

This block uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the references listed below.

Pair Block

Binary-Input RS Encoder

References

[1] Wicker, Stephen B., *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, N.J., Prentice Hall, 1995.

[2] Berlekamp, Elwyn R., *Algebraic Coding Theory*, New York, McGraw-Hill, 1968.

[3] Clark, George C., Jr., and J. Bibb Cain, *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

See Also

Integer-Output RS Decoder

Binary Symmetric Channel

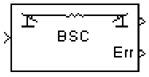
Purpose

Introduce binary errors

Library

Channels

Description



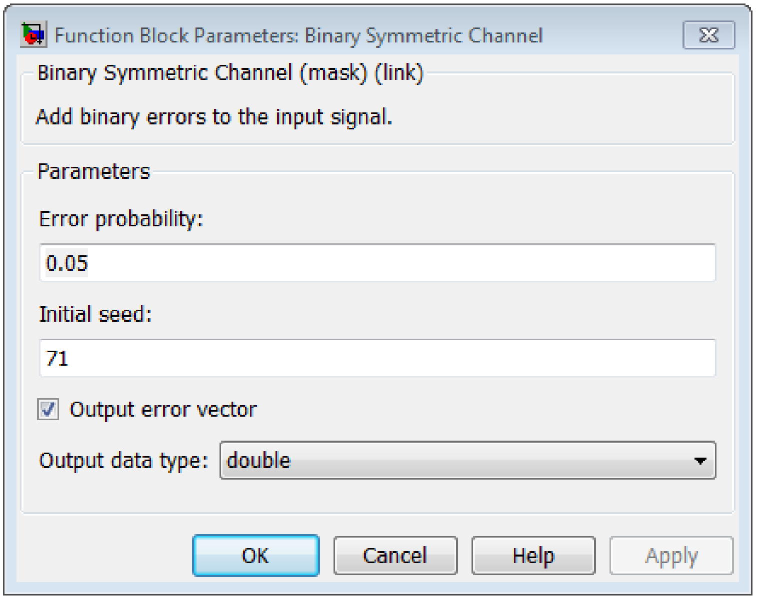
The Binary Symmetric Channel block introduces binary errors to the signal transmitted through this channel.

The input port represents the transmitted binary signal. This block accepts a scalar or vector input signal. The block processes each vector element independently, and introduces an error in a given spot with probability **Error probability**.

This block uses the DSP System Toolbox Random Source block to generate the noise. The block generates random numbers using the Ziggurat method, which is the same method used by the MATLAB `randn` function. The **Initial seed** parameter in this block initializes the noise generator. **Initial seed** can be either a scalar or a vector, with a length that matches the number of channels in the input signal. For details on **Initial seed**, see the Random Source block reference page in the DSP System Toolbox documentation set.

The first output port is the binary signal the channel processes. The second output port is the vector of errors the block introduces. To suppress the second output port, clear **Output error vector**.

Dialog Box



Error probability

The probability that a binary error occurs. Set the value of this parameter between 0 and 1.

Initial seed

The initial seed value for the random number generator.

Output error vector

When you select this box the block outputs the vector of errors.

Output data type

Select the output data type as double or boolean.

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 • Boolean

See Also

Bernoulli Binary Generator

Bipolar to Unipolar Converter

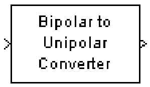
Purpose

Map bipolar signal into unipolar signal in range [0, M-1]

Library

Utility Blocks

Description

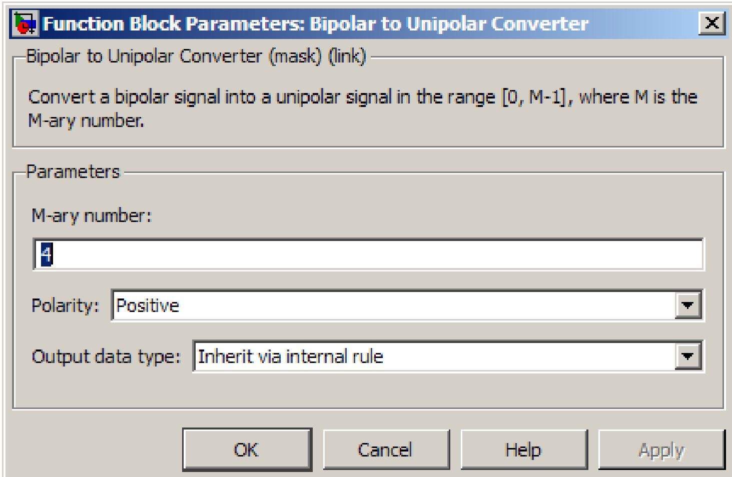


The Bipolar to Unipolar Converter block maps the bipolar input signal to a unipolar output signal. If the input consists of integers in the set {-M+1, -M+3, -M+5,..., M-1}, where M is the **M-ary number** parameter, then the output consists of integers between 0 and M-1. This block is only designed to work when the input value is within the set {-M+1, -M+3, -M+5,..., M-1}, where M is the **M-ary number** parameter. If the input value is outside of this set of integers the output may not be valid.

The table below shows how the block's mapping depends on the **Polarity** parameter.

Polarity Parameter Value	Output Corresponding to Input Value of k
`Positive`	(M-1+k)/2
`Negative`	(M-1-k)/2

Dialog Box



M-ary number

The number of symbols in the bipolar or unipolar alphabet.

Polarity

A value of `Positive` causes the block to maintain the relative ordering of symbols in the alphabets. A value of `Negative` causes the block to reverse the relative ordering of symbols in the alphabets.

Output Data Type

The type of bipolar signal produced at the block's output.

The block supports the following output data types:

- `Inherit via internal rule`
- `Same as input`
- `double`
- `int8`
- `uint8`
- `int16`

- `uint16`
- `int32`
- `uint32`
- `boolean`

When the parameter is set to its default setting, `Inherit via internal rule`, the block determines the output data type based on the input data type.

- If the input signal is floating-point (either `single` or `double`), the output data type is the same as the input data type.
- If the input data type is not floating-point:
  - Based on the **M-ary number** parameter, the output data type is the ideal unsigned integer output word length required to contain the range [0 M-1] and is computed as follows:

    ideal word length = ceil(log2(M))
  - The block sets the output data type to be an unsigned integer, based on the smallest word length (in bits) that can fit best the computed ideal word length.

Note The selections in the **Hardware Implementation** pane pertaining to word length constraints do not affect how this block determines output data types.

Examples

If the input is [-3; -1; 1; 3], the **M-ary number** parameter is `4`, and the **Polarity** parameter is `Positive`, then the output is [0; 1; 2; 3]. Changing the **Polarity** parameter to `Negative` changes the output to [3; 2; 1; 0].

If the value for the **M-ary number** is 2^8 the block gives an output of uint8.

If the value for the **M-ary number** is 2^8+1 the block gives an output of uint16.

Pair Block

Unipolar to Bipolar Converter

Purpose

Map vector of bits to corresponding vector of integers

Library

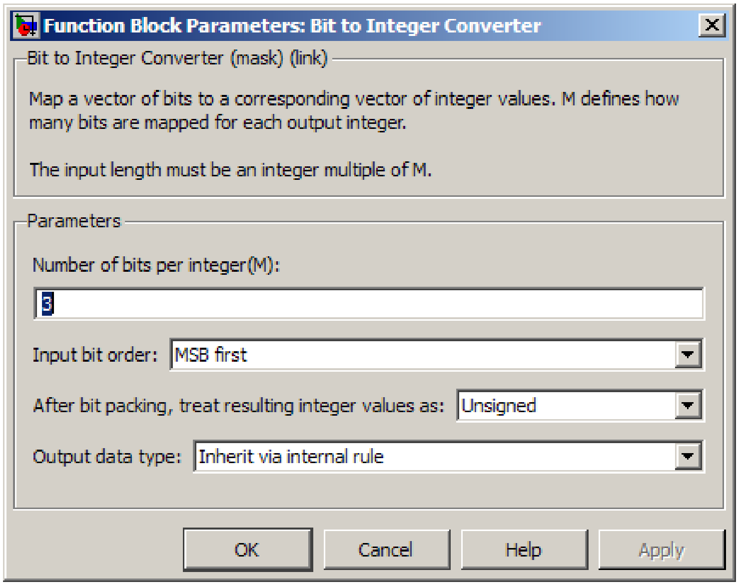
Utility Blocks

Description

The Bit to Integer Converter block maps groups of bits in the input vector to integers in the output vector. M defines how many bits are mapped for each output integer.

For unsigned integers, if M is the **Number of bits per integer**, then the block maps each group of M bits to an integer between 0 and 2^{M}-1. As a result, the output vector length is 1/M times the input vector length. For signed integers, if M is the **Number of bits per integer** , then the block maps each group of M bits to an integer between -2^{M-1} and 2^{M-1}-1.

This block accepts a column vector input signal with an integer multiple equal to the value you specify for **Number of bits per integer** parameter. The block accepts `double`, `single`, `boolean`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32` and `ufix1` input data types.



Dialog Box

Number of bits per integer

The number of input bits that the block maps to each integer of the output. This parameter must be an integer between 1 and 32.

Input bit order

Defines whether the first bit of the input signal is the most significant bit (MSB) or the least significant bit (LSB). The default selection is MSB.

After bit packing, treat resulting integer value as

Indicates if the integer value input ranges should be treated as signed or unsigned. The default setting is `Unsigned`.

Note This parameter setting determines which **Output data type** selections are available.

Output data type

If the input values are unsigned integers, you can choose from the following **Output data type** options:

- `Inherit via internal rule`
- `Smallest integer`
- `Same as input`
- `double`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`

If the input values are signed integers, you can choose from the following **Output data type** options:

- `Inherit via internal rule`
- `Smallest integer`
- `double`
- `single`
- `int8`
- `int16`
- `int32`

The default selection for this parameter is `Inherit via internal rule`.

When you set the parameter to `Inherit via internal rule`, the block determines the output data type based on the input data type.

- If the input signal is floating-point (either `double` or `single`), the output data type is the same as the input data type.
- If the input data type is not floating-point, the output data type is determined as if the parameter is set to `Smallest integer` .

When you set the parameter to `Smallest integer`, the software selects the output data type based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- If `ASIC/FPGA` is selected, the output data type is the smallest ideal integer or fixed-point data type, based on the setting for the **Number of bits per integer** parameter.
- For all other selections, the output data type is the smallest available (signed or unsigned) integer word length that is large enough to fit the ideal minimum bit size.

Examples

Refer to the example on the Integer to Bit Converter reference page: Fixed-Point Integer To Bit and Bit To Integer Conversion (Audio Scrambling and Descrambling Example)

See Also

`bi2de`, `bin2dec`

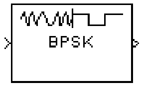
Pair Block

Integer to Bit Converter

Purpose Demodulate BPSK-modulated data

Library PM, in Digital Baseband sublibrary of Modulation

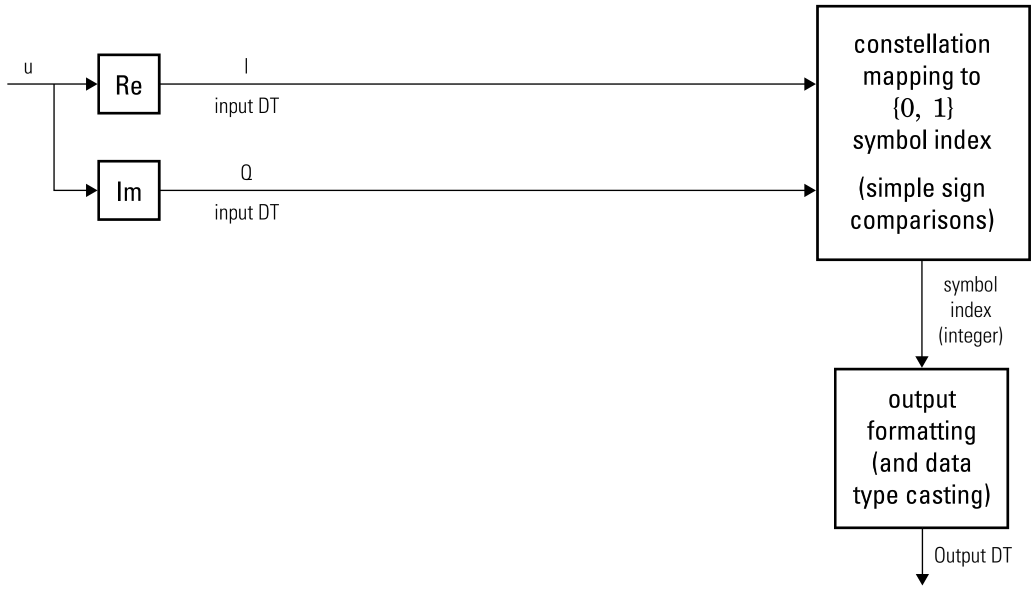
Description



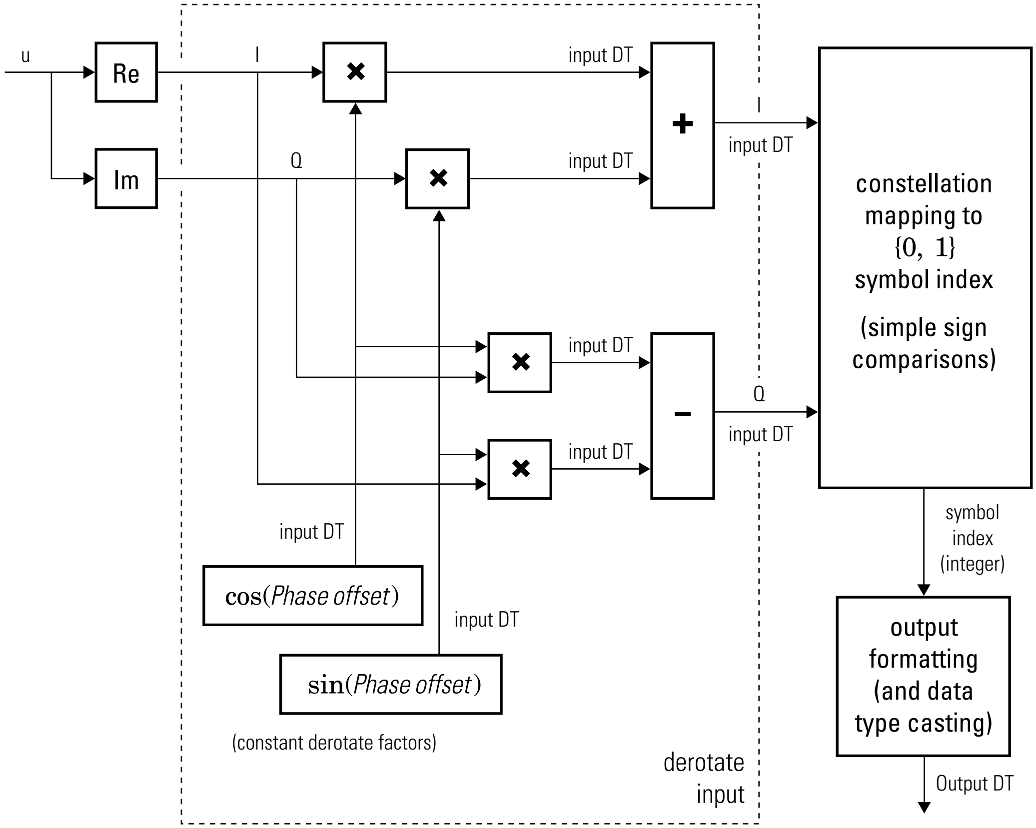
The BPSK Demodulator Baseband block demodulates a signal that was modulated using the binary phase shift keying method. The input is a baseband representation of the modulated signal. This block accepts a scalar or column vector input signal. The input signal must be be a discrete-time complex signal. The block maps the points exp(jθ) and -exp(jθ) to 0 and 1, respectively, where θ is the **Phase offset** parameter.

For information about the data types each block port supports, see "Supported Data Types" on page 2-105.

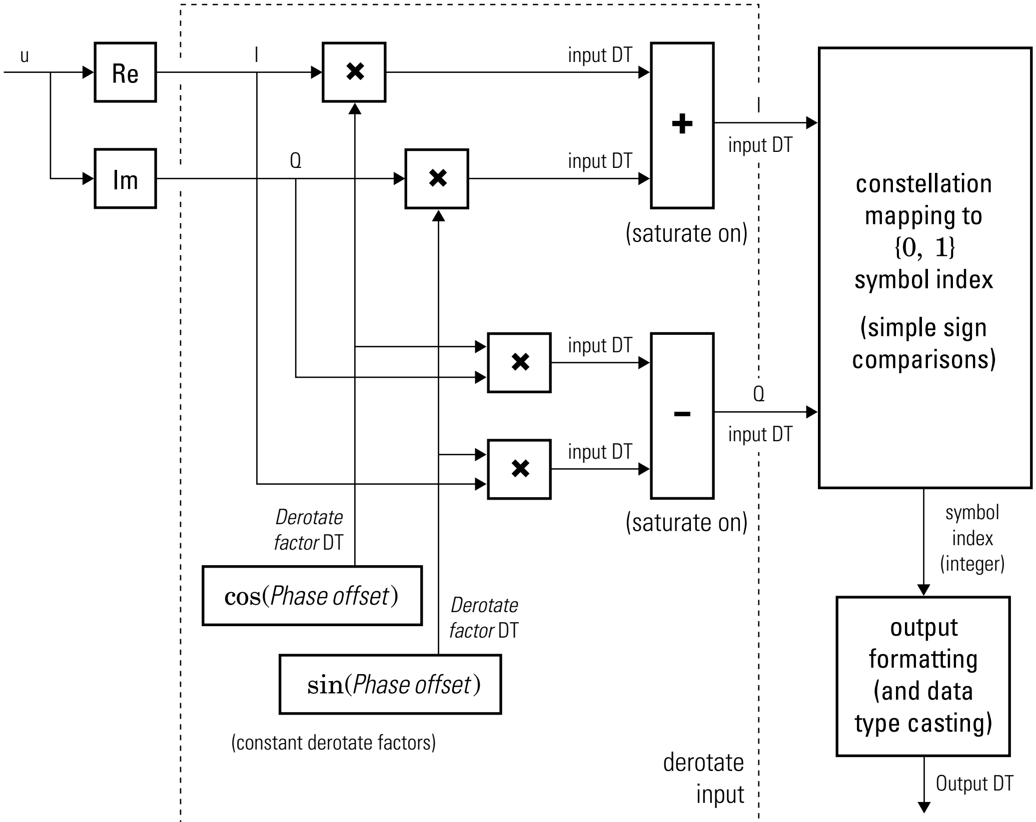
Algorithm



Hard-Decision BPSK Demodulator Signal Diagram for Trivial Phase Offset (multiple of $\frac{\pi}{2}$)



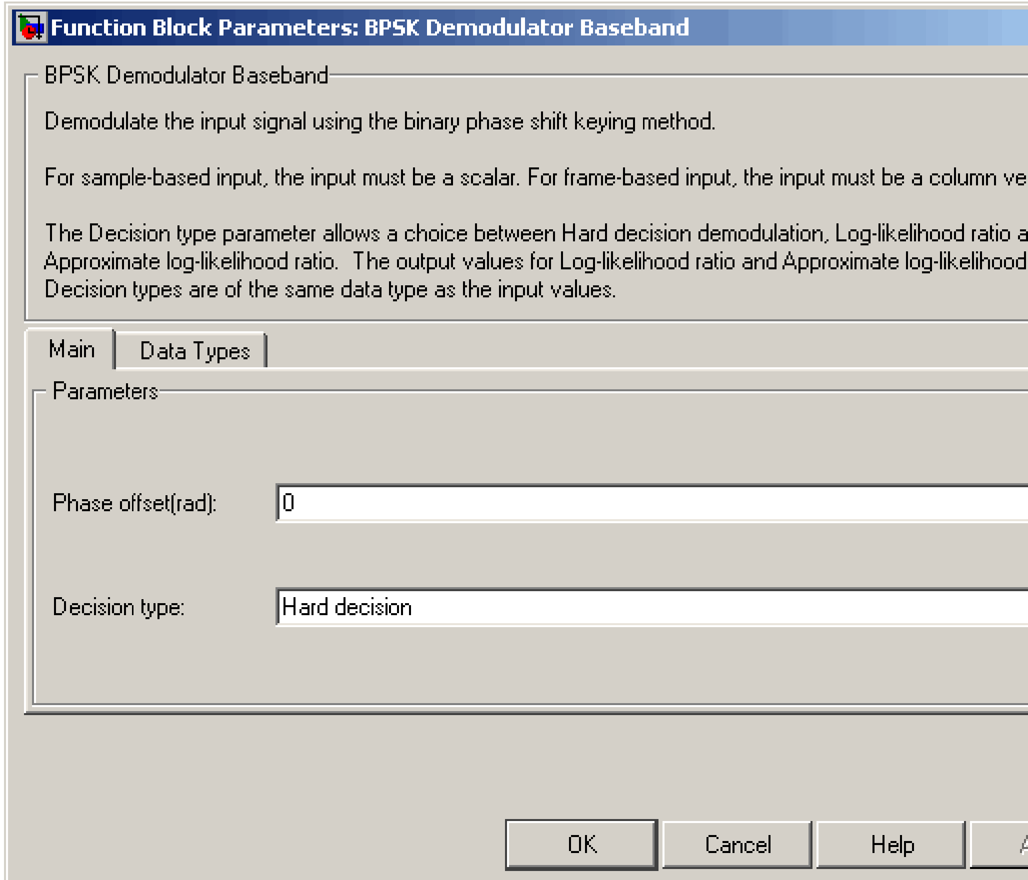
Hard-Decision BPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



Hard-Decision BPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

The exact LLR and approximate LLR cases (soft-decision) are described in “Exact LLR Algorithm” and “Approximate LLR Algorithm” in the *Communications System Toolbox User’s Guide*.

Dialog Box



Phase offset (rad)
: The phase of the zeroth point of the signal constellation.

Decision type
: Specifies the use of hard decision, LLR, or approximate LLR during demodulation. The output values for Log-likelihood ratio and Approximate log-likelihood ratio are of the same data type as the input values. See “Exact LLR Algorithm” and “Approximate

LLR Algorithm” in the *Communications System Toolbox User’s Guide* for algorithm details.

Noise variance source

This field appears when `Approximate log-likelihood ratio` or `Log-likelihood ratio` is selected for **Decision type**.

When set to `Dialog`, the noise variance can be specified in the **Noise variance** field. When set to `Port`, a port appears on the block through which the noise variance can be input.

Noise variance

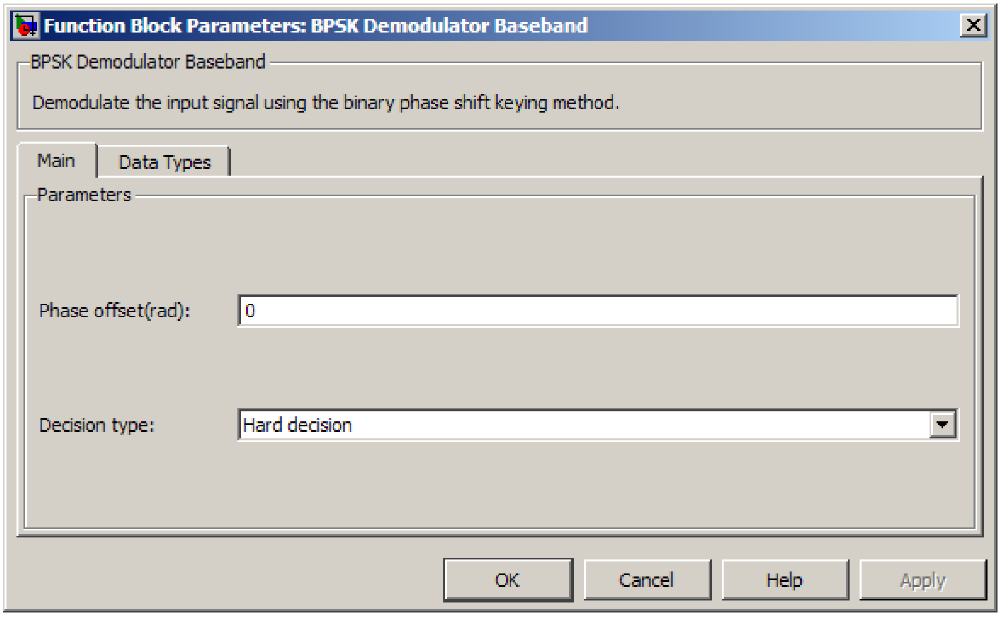
This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

The LLR algorithm involves computing exponentials of very large or very small numbers using finite precision arithmetic and would yield:

- `Inf` to `-Inf` if **Noise variance** is very high
- `NaN` if **Noise variance** and signal power are both very small

In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.



Data Types Pane for Hard-Decision

Output

When **Decision type** is set to `Hard decision`, the output data type can be set to `'Inherit via internal rule'`, `'Smallest unsigned integer'`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `boolean`.

When this parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is a floating-point type (`single` or `double`). If the input data type is fixed-point, the output data type will work as if this parameter is set to `'Smallest unsigned integer'`.

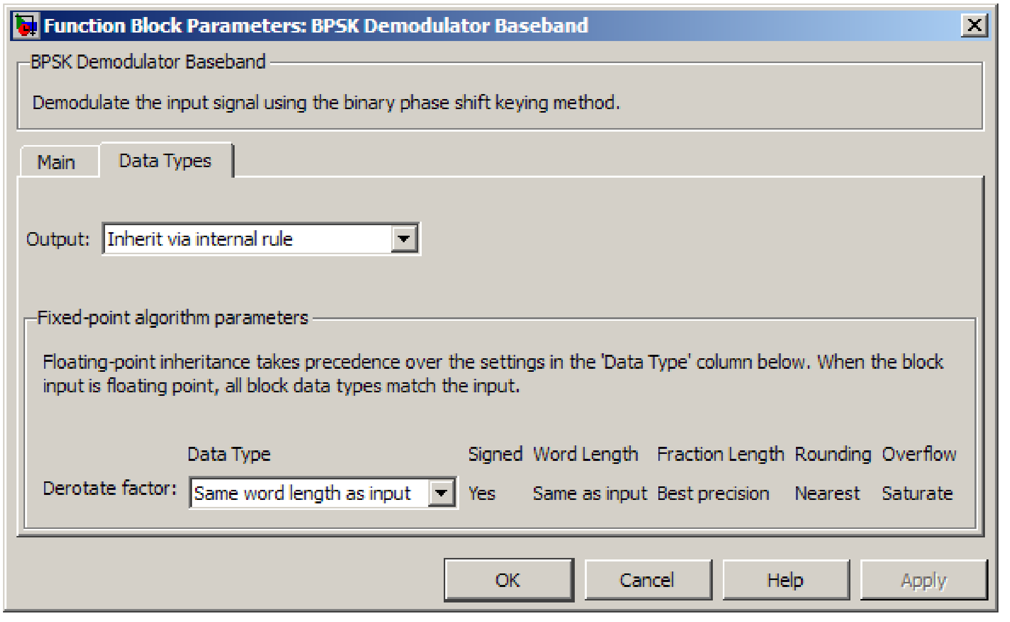
When this parameter is set to `'Smallest unsigned integer'`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration

Parameters dialog box of the model. If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type is the ideal minimum one-bit size, i.e., `ufix(1)`. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding to the size of a char (e.g., `uint8`).

Derotate factor

This parameter only applies when the input is fixed-point and **Phase offset** is not a multiple of $\frac{\pi}{2}$.

This can be set to `Same word length as input` or `Specify word length`, in which case a field is enabled for user input.



Data Types Pane for Soft-Decision

When **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the output data type is inherited from the input (e.g., if the input is of data type `double`, the output is also of data type `double`).

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed point (only for `Hard decision` mode)
Var	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) in `ASIC/FPGA` and when **Decision type** is `Hard decision` modes

Pair Block

BPSK Modulator Baseband

See Also

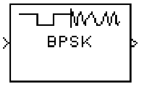
M-PSK Demodulator Baseband, QPSK Demodulator Baseband, DBPSK Demodulator Baseband

BPSK Modulator Baseband

Purpose Modulate using binary phase shift keying method

Library PM, in Digital Baseband sublibrary of Modulation

Description The BPSK Modulator Baseband block modulates using the binary phase shift keying method. The output is a baseband representation of the modulated signal.

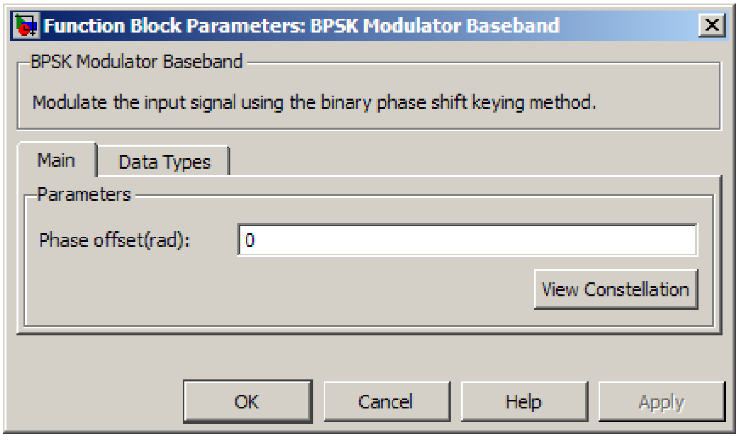


This block accepts a column vector input signal. The input must be a discrete-time binary-valued signal. If the input bit is 0 or 1, respectively, then the modulated symbol is exp(jθ) or -exp(jθ), respectively, where θ represents the **Phase offset** parameter.

For information about the data types each block port supports, see the "Supported Data Types" on page 2-109 table on this page.

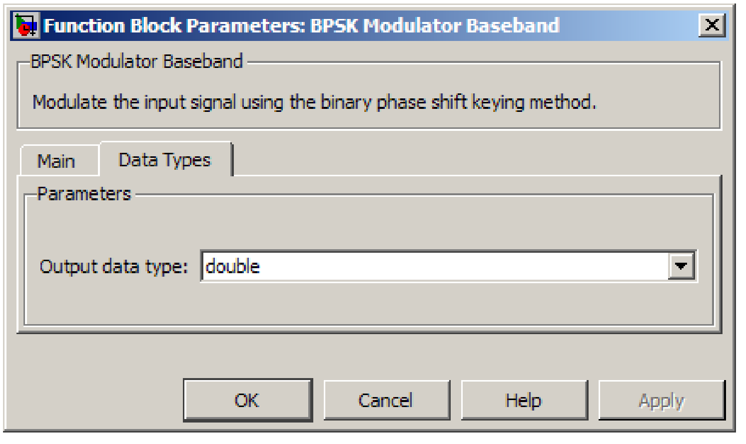
Constellation Visualization

The BPSK Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. For more information, see the "Constellation Visualization" section of the *Communications System Toolbox User's Guide*.



Dialog Box

Phase offset (rad)
The phase of the zeroth point of the signal constellation.



Output data type
The output data type can be set to `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

Setting this parameter to `Fixed-point` or `User-defined` enables fields in which you can further specify details. Setting this parameter to `Inherit via back propagation`, sets the output data type and scaling to match the following block.

Output 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.

Set output fraction length 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 **Output 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.

User-defined data type

Specify any signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer™. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Output 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 output fraction length to** parameter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1)
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only)

Pair Block

BPSK Demodulator Baseband

See Also

M-PSK Modulator Baseband, QPSK Modulator Baseband, DBPSK Modulator Baseband

Charge Pump PLL

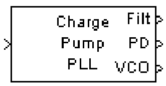
Purpose

Implement charge pump phase-locked loop using digital phase detector

Library

Components sublibrary of Synchronization

Description



The Charge Pump PLL (phase-locked loop) block automatically adjusts the phase of a locally generated signal to match the phase of an input signal. It is suitable for use with digital signals.

This PLL has these three components:

- A sequential logic phase detector, also called a digital phase detector or a phase/frequency detector.
- A filter. You specify the filter transfer function using the **Lowpass filter numerator** and **Lowpass filter denominator** parameters. Each is a vector that gives the respective polynomial's coefficients in order of descending powers of s.

  To design a filter, use functions such as `butter`, `cheby1`, and `cheby2` in Signal Processing Toolbox software. The default filter is a Chebyshev type II filter whose transfer function arises from the command below.

  ```
  [num, den] = cheby2(3,40,100,'s')
  ```

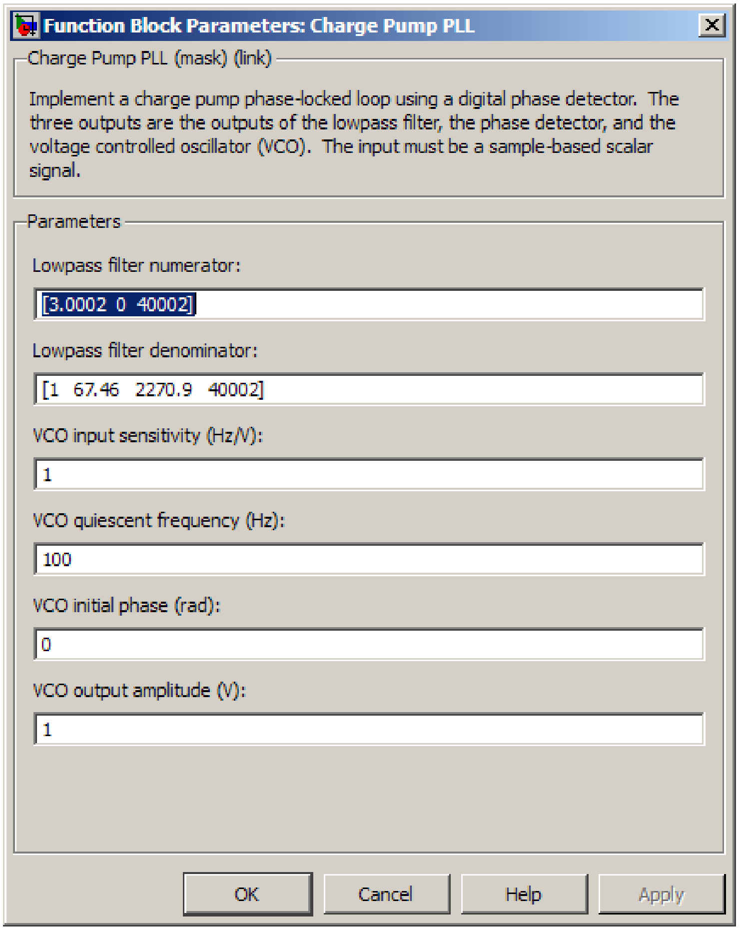
- A voltage-controlled oscillator (VCO). You specify characteristics of the VCO using the **VCO input sensitivity**, **VCO quiescent frequency**, **VCO initial phase**, and **VCO output amplitude** parameters.

This block accepts a sample-based scalar input signal. The input signal represents the received signal. The three output ports produce:

- The output of the filter
- The output of the phase detector
- The output of the VCO

A sequential logic phase detector operates on the zero crossings of the signal waveform. The equilibrium point of the phase difference between the input signal and the VCO signal equals π. The sequential logic detector can compensate for any frequency difference that might exist between a VCO and an incoming signal frequency. Hence, the sequential logic phase detector acts as a frequency detector.

Dialog Box



Lowpass filter numerator

The numerator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of s.

Lowpass filter denominator
The denominator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of s.

VCO input sensitivity (Hz/V)
This value scales the input to the VCO and, consequently, the shift from the **VCO quiescent frequency** value. The units of **VCO input sensitivity** are Hertz per volt.

VCO quiescent frequency (Hz)
The frequency of the VCO signal when the voltage applied to it is zero. This should match the frequency of the input signal.

VCO initial phase (rad)
The initial phase of the VCO signal.

VCO output amplitude
The amplitude of the VCO signal.

See Also

Phase-Locked Loop

References

For more information about digital phase-locked loops, see the works listed in“Selected Bibliography for Synchronization” in *Communications System Toolbox User's Guide*.

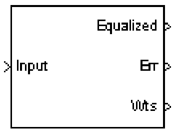
Purpose

Equalize using constant modulus algorithm

Library

Equalizers

Description



The CMA Equalizer block uses a linear equalizer and the constant modulus algorithm (CMA) to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the CMA to update the weights, once per symbol. If the **Number of samples per symbol** parameter is `1`, then the block implements a symbol-spaced equalizer; otherwise, the block implements a fractionally spaced equalizer.

When using this block, you should initialize the equalizer weights with a nonzero vector. Typically, CMA is used with differential modulation; otherwise, the initial weights are very important. A typical vector of initial weights has a 1 corresponding to the center tap and zeros elsewhere.

Input and Output Signals

The `Input` port accepts a scalar-valued or column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal.

You can configure the block to have one or more of the extra ports listed in the table below.

Port	Meaning	How to Enable
`Err` output	$y(R - \|y\|^2)$, where y is the equalized signal and R is a constant related to the signal constellation	Select **Output error**.
`Wts` output	A vector listing the weights after the block has processed either the current	Select **Output weights**.

Port	Meaning	How to Enable
	input frame or sample.	

Algorithms

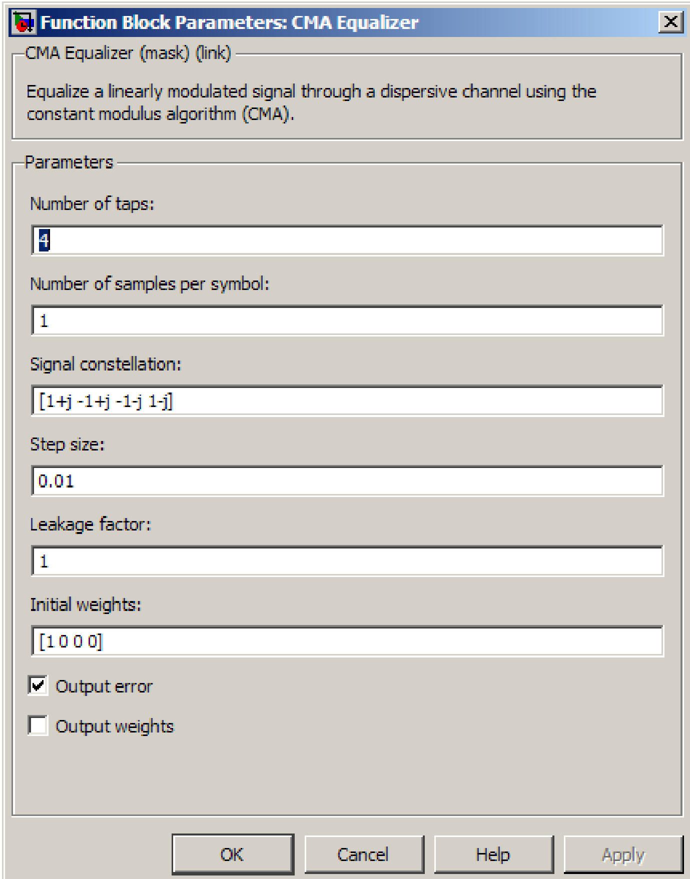
Referring to the schematics in "Equalizer Structure", define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of weights, w, this adaptive algorithm creates the new set of weights given by

$$(\texttt{LeakageFactor})\ w + (\texttt{StepSize})\ u^* e$$

where the * operator denotes the complex conjugate.

Equalizer Delay

The delay between the transmitter's modulator output and the CMA equalizer output is typically unknown (unlike the delay for other adaptive equalizers in this product). If you need to determine the delay, you can use the Find Delay block.



Dialog Box

Number of taps

The number of taps in the filter of the equalizer.

Number of samples per symbol

The number of input samples for each symbol.

When you set this parameter to 1, the filter weights are updated once for each symbol, for a symbol spaced (i.e. T-spaced) equalizer. When you set this parameter to a value greater than one, the weights are updated once every N^{th} sample, for a fractionally spaced (i.e. T/N-spaced) equalizer.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulation.

Step size
The step size of the CMA.

Leakage factor
The leakage factor of the CMA, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights
A vector that lists the initial weights for the taps.

Output error
If you check this box, the block outputs the error signal described in the table above.

Output weights
If you check this box, the block outputs the current weights.

References

[1] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[2] Johnson, Richard C. Jr., Philip Schniter, Thomas. J. Endres, et al., "Blind Equalization Using the Constant Modulus Criterion: A Review," *Proceedings of the IEEE*, vol. 86, pp. 1927-1950, October 1998.

See Also

LMS Linear Equalizer, LMS Decision Feedback Equalizer, RLS Linear Equalizer, RLS Decision Feedback Equalizer

Complex Phase Difference

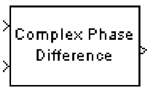
Purpose

Output phase difference between two complex input signals

Library

Utility Blocks

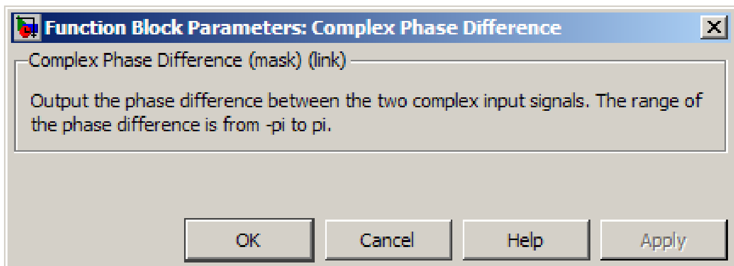
Description



The Complex Phase Difference block accepts two complex input signals that have the same size and frame status. The output is the phase difference from the second to the first, measured in radians. The elements of the output are between -π and π.

The input signals can have any size or frame status. This block processes each pair of elements independently.

Dialog Box



See Also

Complex Phase Shift

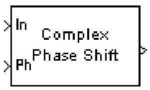
Purpose

Shift phase of complex input signal by second input value

Library

Utility Blocks

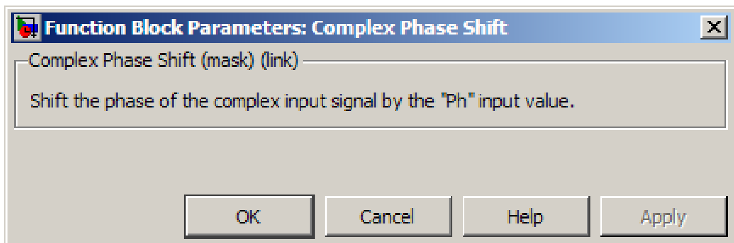
Description



The Complex Phase Shift block accepts a complex signal at the port labeled `In`. The output is the result of shifting this signal's phase by an amount specified by the real signal at the input port labeled `Ph`. The `Ph` input is measured in radians, and must have the same size and frame status as the `In` input.

The input signals can have any size or frame status. This block processes each pair of corresponding elements independently.

Dialog Box



See Also

Complex Phase Difference

Continuous-Time VCO

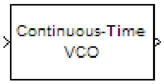
Purpose

Implement voltage-controlled oscillator

Library

Components sublibrary of Synchronization

Description



Continuous-Time VCO

The Continuous-Time VCO (voltage-controlled oscillator) block generates a signal with a frequency shift from the **Quiescent frequency** parameter that is proportional to the input signal. The input signal is interpreted as a voltage. If the input signal is $u(t)$, then the output signal is

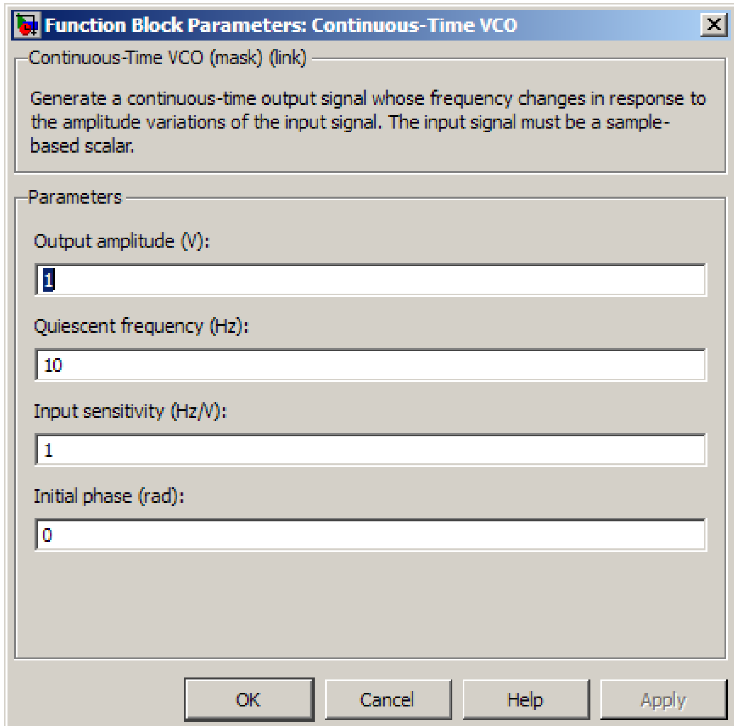
$$y(t) = A_c \cos\left(2\pi f_c t + 2\pi k_c \int_0^t u(\tau)d\tau + \varphi\right)$$

where A_c is the **Output amplitude** parameter, f_c is the **Quiescent frequency** parameter, k_c is the **Input sensitivity** parameter, and φ is the **Initial phase** parameter.

This block uses a continuous-time integrator to interpret the equation above.

The input and output are both sample-based scalar signals.

Dialog Box



Output amplitude
: The amplitude of the output.

Quiescent frequency
: The frequency of the oscillator output when the input signal is zero.

Input sensitivity
: This value scales the input voltage and, consequently, the shift from the **Quiescent frequency** value. The units of **Input sensitivity** are Hertz per volt.

Initial phase
: The initial phase of the oscillator in radians.

See Also

Discrete-Time VCO

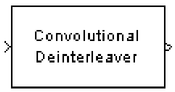
Purpose

Restore ordering of symbols that were permuted using shift registers

Library

Convolutional sublibrary of Interleaving

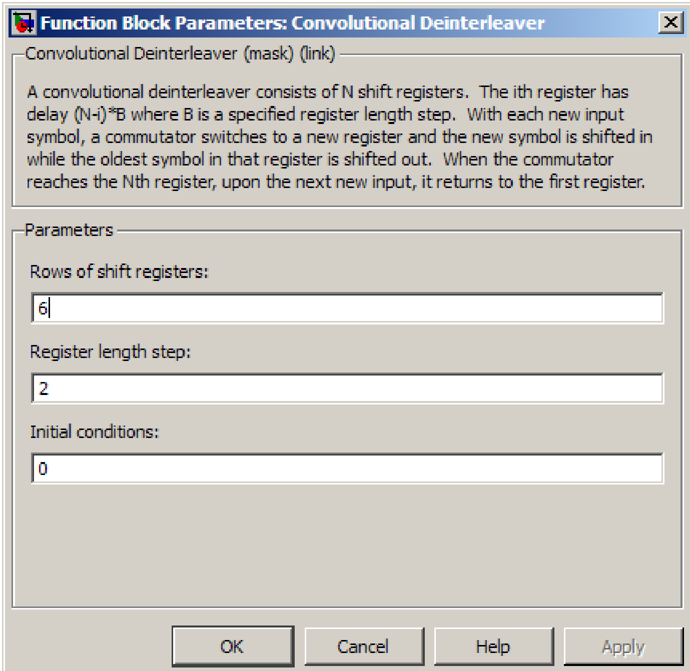
Description



The Convolutional Deinterleaver block recovers a signal that was interleaved using the Convolutional Interleaver block. Internally, this block uses a set of shift registers. The delay value of the k^{th} shift register is (N-k) times the **Register length step** parameter. The number of shift registers, N, is the value of the **Rows of shift registers** parameter. The parameters in the two blocks must have the same values.

This block accepts a scalar or column vector input signal, which can be real or complex. The output signal has the same sample time as the input signal.

This block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point.



Dialog Box

Rows of shift registers

The number of shift registers that the block uses internally.

Register length step

The difference in symbol capacity of each successive shift register, where the last register holds zero symbols.

Initial conditions

Indicates the values that fill each shift register at the beginning of the simulation (except for the last shift register, which has zero delay).

- When you select a scalar value for **Initial conditions**, the value fills all shift registers (except for the last one)

- When you select a column vector with a length equal to the **Rows of shift registers** parameter, each entry fills the corresponding shift register.

The value of the first element of the **Initial conditions** parameter is unimportant, since the last shift register has zero delay.

Examples

For an example that uses this block, see “Adaptive Algorithms”.

Pair Block

Convolutional Interleaver

See Also

General Multiplexed Deinterleaver, Helical Deinterleaver

References

[1] Clark, George C. Jr. and J. Bibb Cain. *Error-Correction Coding for Digital Communications*. New York: Plenum Press, 1981.

[2] Forney, G., D., Jr. "Burst-Correcting Codes for the Classic Bursty Channel." *IEEE Transactions on Communications*, vol. COM-19, October 1971. 772-781.

[3] Ramsey, J. L. "Realization of Optimum Interleavers." *IEEE Transactions on Information Theory*, IT-16 (3), May 1970. 338-345.

Convolutional Encoder

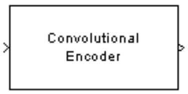
Purpose

Create convolutional code from binary data

Library

Convolutional sublibrary of Error Detection and Correction

Description



The Convolutional Encoder block encodes a sequence of binary input vectors to produce a sequence of binary output vectors. This block can process multiple symbols at a time.

This block can accept inputs that vary in length during simulation. For more information about variable-size signals, see “Variable-Size Signal Basics” in the Simulink documentation.

Input and Output Sizes

If the encoder takes k input bit streams (that is, it can receive 2^k possible input symbols), the block input vector length is L*k for some positive integer L. Similarly, if the encoder produces n output bit streams (that is, it can produce 2^n possible output symbols), the block output vector length is L*n.

This block accepts a column vector input signal with any positive integer for L. For variable-size inputs, the L can vary during simulation. The operation of the block is governed by the **Operation mode** parameter.

For both its inputs and outputs for the data ports, the block supports `double`, `single`, `boolean`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, and `ufix1`. The port data types are inherited from the signals that drive the block. The input reset port supports `double` and `boolean` typed signals.

Specifying the Encoder

To define the convolutional encoder, use the **Trellis structure** parameter. This parameter is a MATLAB structure whose format is described in the “Trellis Description of a Convolutional Code” section of the Communications System Toolbox documentation. You can use this parameter field in two ways:

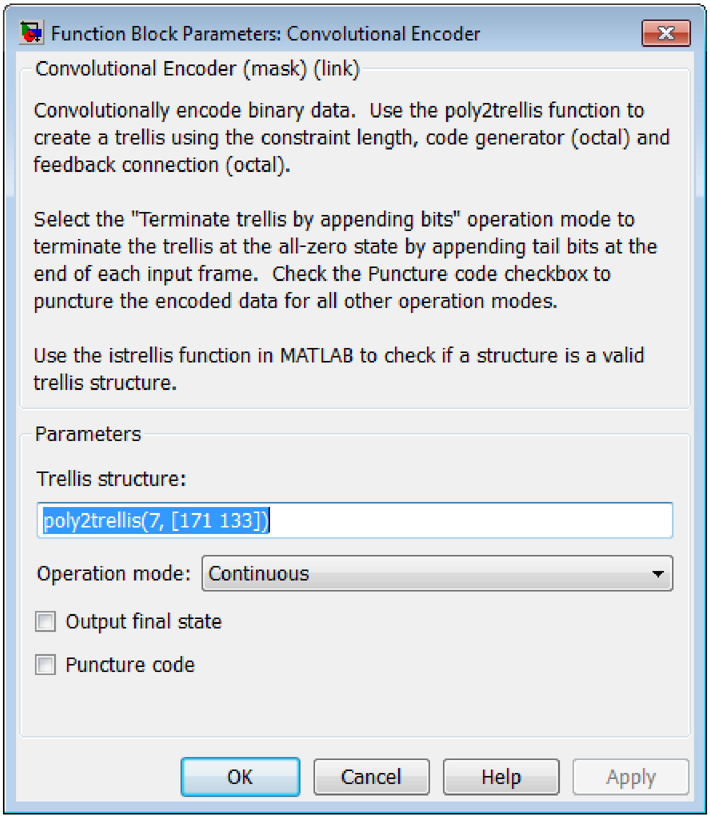
- If you have a variable in the MATLAB workspace that contains the trellis structure, enter its name in the **Trellis structure** parameter.

This way is preferable because it causes Simulink to spend less time updating the diagram at the beginning of each simulation, compared to the usage described next.

- If you want to specify the encoder using its constraint length, generator polynomials, and possibly feedback connection polynomials, use a `poly2trellis` command in the **Trellis structure** parameter. For example, to use an encoder with a constraint length of 7, code generator polynomials of 171 and 133 (in octal numbers), and a feedback connection of 171 (in octal), set the **Trellis structure** parameter to

  ```
  poly2trellis(7,[171 133],171)
  ```

The encoder registers begin in the all-zeros state. Set the **Operation mode** parameter to `Reset on nonzero input via port` to reset all encoder registers to the all-zeros state during the simulation. This selection opens a second input port, labeled `Rst`, which accepts a scalar-valued input signal. When the input signal is nonzero, the block resets before processing the data at the first input port. To reset the block after it processes the data at the first input port, select **Delay reset action to next time step**.



Dialog Box

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Operation mode

In `Continuous` mode, the block retains the encoder states at the end of each input, for use with the next frame.

In `Truncated (reset every frame)` mode, the block treats each input independently. The encoder states are reset to all-zeros state at the start of each input.

Note When this block outputs sequences that vary in length during simulation and you set the **Operation mode** to `Truncated (reset every frame)` or `Terminate trellis by appending bits`, the block's state resets at every input time step.

In `Terminate trellis by appending bits` mode, the block treats each input independently. For each input frame, extra bits are used to set the encoder states to all-zeros state at the end of the frame. The output length is given by

$y = n \cdot (x + s) / k$, where x is the number of input bits, and

$s = \text{constraint length} - 1$ (or, in the case of multiple constraint lengths, s =`sum(ConstraintLength(i)-1)`).

Note This block works for cases $k \geq 1$, where it has the same values for constraint lengths in each input stream (e.g., constraint lengths of [2 2] or [7 7] will work, but [5 4] will not).

In `Reset on nonzero input via port` mode, the block has an additional input port, labeled `Rst`. When the `Rst` input is nonzero, the encoder resets to the all-zeros state.

Delay reset action to next time step

When you select **Delay reset action to next time step**, the Convolutional Encoder block resets after computing the encoded data. This check box only appears when you set the **Operation mode** parameter to `Reset on nonzero input via port`.

The delay in the reset action allows the block to support HDL code generation. In order to generate HDL code, you must have an HDL Coder™ license.

Output final state

When you select **Output final state**, the second output port signal specifies the output state for the block. The output signal is a scalar, integer value. You can select **Output final state** for all operation modes except `Terminate trellis by appending bits` .

Specify initial state via input port

When you select **Specify initial state via input port** the second input port signal specifies the starting state for every frame in the block. The input signal must be a scalar, integer value. **Specify initial state via input port** appears only in `Truncated` operation mode.

Puncture code

Selecting this option opens the field **Puncture vector**.

Puncture vector

Vector used to puncture the encoded data. The puncture vector is a pattern of `1`s and `0`s where the `0`s indicate the punctured bits. This field appears when you select **Punctured code**.

Puncture Pattern Examples

For some commonly used puncture patterns for specific rates and polynomials, see the last three references listed on this page.

See Also

Viterbi Decoder, APP Decoder

References

[1] Clark, George C. Jr. and J. Bibb Cain, *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

[2] Gitlin, Richard D., Jeremiah F. Hayes, and Stephen B. Weinstein, *Data Communications Principles*, New York, Plenum, 1992.

[3] Yasuda, Y., et. al., “High rate punctured convolutional codes for soft decision Viterbi decoding,” *IEEE Transactions on Communications*, Vol. COM-32, No. 3, pp 315–319, March 1984.

[4] Haccoun, D., and Begin, G., “High-rate punctured convolutional codes for Viterbi and Sequential decoding,” *IEEE Transactions on Communications*, Vol. 37, No. 11, pp 1113–1125, Nov. 1989.

[5] Begin, G., et.al., “Further results on high-rate punctured convolutional codes for Viterbi and sequential decoding,” *IEEE Transactions on Communications*, Vol. 38, No. 11, pp 1922–1928, Nov. 1990.

Purpose

Permute input symbols using set of shift registers

Library

Convolutional sublibrary of Interleaving

Description

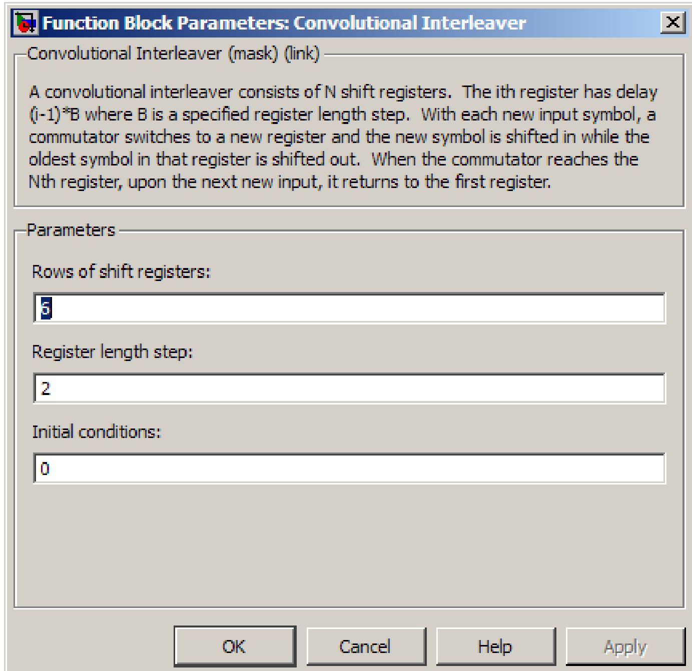
The Convolutional Interleaver block permutes the symbols in the input signal. Internally, it uses a set of shift registers. The delay value of the kth shift register is (k-1) times the **Register length step** parameter. The number of shift registers is the value of the **Rows of shift registers** parameter.

The **Initial conditions** parameter indicates the values that fill each shift register at the beginning of the simulation (except for the first shift register, which has zero delay). If **Initial conditions** is a scalar, then its value fills all shift registers except the first; if **Initial conditions** is a column vector whose length is the **Rows of shift registers** parameter, then each entry fills the corresponding shift register. The value of the first element of the **Initial conditions** parameter is unimportant, since the first shift register has zero delay.

This block accepts a scalar or column vector input signal, which can be real or complex. The output signal has the same sample time as the input signal.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The data type of this output will be the same as that of the input signal.

Dialog Box



Rows of shift registers
: The number of shift registers that the block uses internally.

Register length step
: The number of additional symbols that fit in each successive shift register, where the first register holds zero symbols.

Initial conditions
: The values that fill each shift register when the simulation begins.

Examples

For an example that uses this block, see "Convolutional Interleaving".

Pair Block

Convolutional Deinterleaver

See Also

General Multiplexed Interleaver, Helical Interleaver

References

[1] Clark, George C. Jr. and J. Bibb Cain. *Error-Correction Coding for Digital Communications*. New York: Plenum Press, 1981.

[2] Forney, G., D., Jr. "Burst-Correcting Codes for the Classic Bursty Channel." *IEEE Transactions on Communications*, vol. COM-19, October 1971. 772-781.

[3] Ramsey, J. L. "Realization of Optimum Interleavers." *IEEE Transactions on Information Theory*, IT-16 (3), May 1970. 338-345.

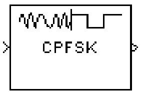
Purpose

Demodulate CPFSK-modulated data

Library

CPM, in Digital Baseband sublibrary of Modulation

Description



The CPFSK Demodulator Baseband block demodulates a signal that was modulated using the continuous phase frequency shift keying method. The input to this block is a baseband representation of the modulated signal. The **M-ary number** parameter, M, is the size of the input alphabet. M must have the form 2^K for some positive integer K.

This block supports multi-h **Modulation index**. See CPM Modulator Baseband for details.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Output type** parameter to `Integer`, then the block produces odd integers between -(M-1) and M-1.

When you set the **Output type** parameter to `Bit`, then the block produces groupings of K bits. Each grouping is called a binary *word*.

In binary output mode, the block first maps each input symbol to an intermediate value as in the integer output mode. The block then maps the odd integer k to the nonnegative integer (k+M-1)/2. Finally, the block maps each nonnegative integer to a binary word, using a mapping that depends on whether the **Symbol set ordering** parameter is set to `Binary` or `Gray`.

This block accepts a scalar-valued or column vector input signal with a data type of `single` or `double`.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input width must be an integer multiple of the **Samples per symbol** parameter value, and the input can be a column vector.

- When you set **Output type** to `Bit`, the output width is K times the number of input symbols.

- When you set **Output type** to `Integer`, the output width is the number of input symbols.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is the product of the input sample time and the **Samples per symbol** parameter value.

- When you set **Output type** to `Bit`, the output width equals the number of bits per symbol.
- When you set **Output type** to `Integer`, the output is a scalar.

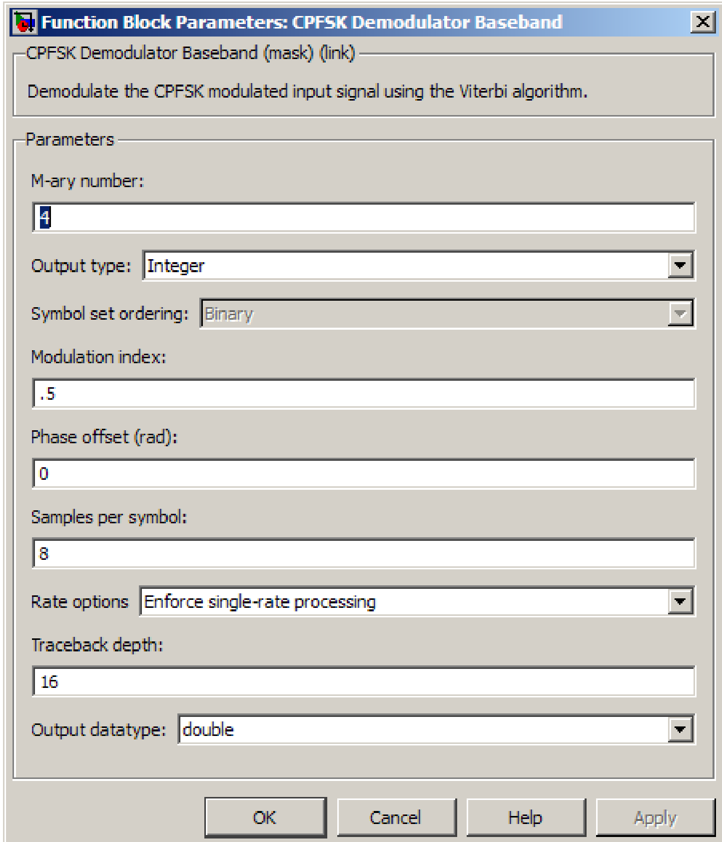
Traceback Depth and Output Delays

Internally, this block creates a trellis description of the modulation scheme and uses the Viterbi algorithm. The **Traceback depth** parameter, *D*, in this block is the number of trellis branches that the algorithm uses to construct each traceback path. *D* influences the output delay, which is the number of zero symbols that precede the first meaningful demodulated value in the output.

- When you set the **Rate options** parameter to `Allow multirate processing`, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking`, then the delay consists of D+1 zero symbols.
- When you set the **Rate options** parameter to `Enforce single-rate processing`, then the delay consists of D zero symbols.

The optimal **Traceback depth** parameter value is dependent on minimum squared Euclidean distance calculations. Alternatively, a typical value, dependent on the number of states, can be chosen using the “five-times-the-constraint-length” rule, which corresponds to $5 \cdot \log2(numStates)$.

For the definition of the number of states, see CPM Demodulator Baseband Help page.



Dialog Box

M-ary number

The size of the alphabet.

Output type

Determines whether the output consists of integers or groups of bits.

Symbol set ordering

Determines how the block maps each integer to a group of output bits. This field is active only when **Output type** is set to `Bit`.

Modulation index

Specify the modulation index $\{h_i\}$. The default is `0.5`. The value of this property must be a real, nonnegative scalar or column vector.

This block supports multi-h **Modulation index**. See CPM Modulator Baseband for details.

Phase offset (rad)

The initial phase of the modulated waveform.

Samples per symbol

The number of input samples that represent each modulated symbol, which must be a positive integer. For more information, see "Upsample Signals and Rate Changes" in *Communications System Toolbox User's Guide*.

Rate options

Select the rate processing method for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width is the number of symbols (which is given by dividing the input length by the **Samples per symbol** parameter value when the **Output type** parameter is set to `Integer`).
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output period is the same as the symbol period and equals the product of the input period and the **Samples per symbol** parameter value.

> **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 *Communications System Toolbox Release Notes* for more information.

For more information, see Single-Rate Processing and Multirate Processing in the Description section of this page.

Traceback depth
The number of trellis branches that the CPFSK Demodulator Baseband block uses to construct each traceback path.

Output datatype
The output data type can be `boolean`, `int8`, `int16`, `int32`, or `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Boolean (When **Output type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (When **Output type** set to `Integer`)

Pair Block

CPFSK Modulator Baseband

See Also

CPM Demodulator Baseband, Viterbi Decoder, M-FSK Demodulator Baseband

References

[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg. *Digital Phase Modulation*. New York: Plenum Press, 1986.

CPFSK Modulator Baseband

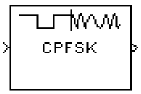
Purpose

Modulate using continuous phase frequency shift keying method

Library

CPM, in Digital Baseband sublibrary of Modulation

Description



The CPFSK Modulator Baseband block modulates a signal using the continuous phase frequency shift keying method. The output is a baseband representation of the modulated signal. The **M-ary number** parameter, M, represents the size of the input alphabet. M must have the form 2^K for some positive integer K.

This block supports multi-h **Modulation index**. See CPM Modulator Baseband for details.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, the block accepts odd integers between -(M-1) and M-1.

When you set the **Input type** parameter to `Bit`, the block accepts groupings of K bits. Each grouping is called a binary *word*. The input vector length must be an integer multiple of K.

In binary input mode, the block maps each binary word to an integer between 0 and M-1, using a mapping scheme that depends on whether you set the **Symbol set ordering** parameter to `Binary` or `Gray`. The block then maps the integer k to the intermediate value 2k-(M-1) and proceeds as if it operates in the integer input mode. For more information, see "Integer-Valued Signals and Binary-Valued Signals" in *Communications System Toolbox User's Guide*.

This block accepts a scalar-valued or column vector input signal. If you set **Input type** to `Bit`, then the input signal can also be a vector of length K.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. In this mode, the input to the block can be multiple symbols.

- When you set **Input type** to `Integer`, the input can be a column vector, the length of which is the number of input symbols.
- When you set **Input type** to `Bit`, the input must be a column vector with a width that is an integer multiple of K, the number of bits per symbol.

The output width equals the product of the number of input symbols and the **Samples per symbol** parameter value.

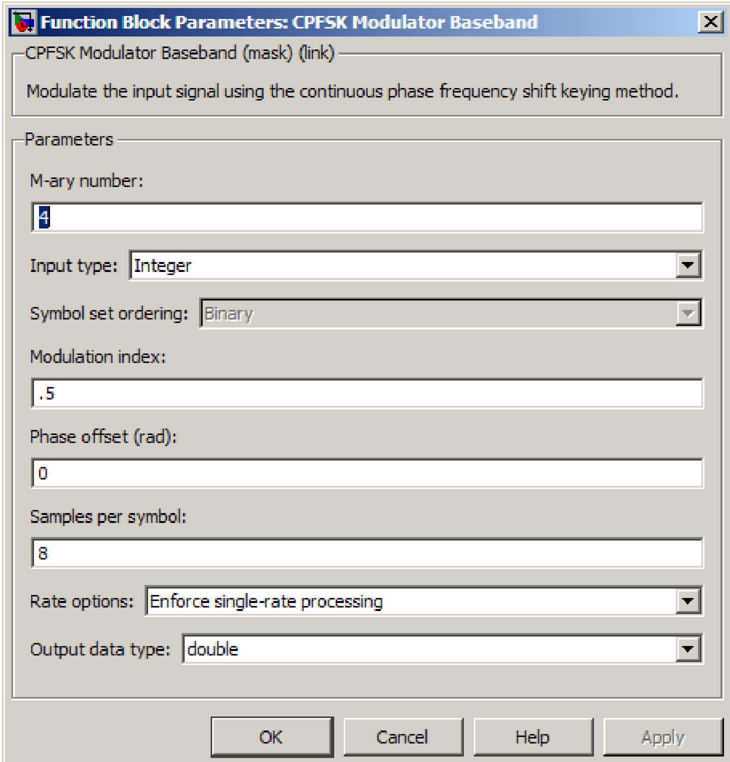
Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

- When you set **Input type** to `Integer`, the input must be a scalar.
- When you set **Input type** to `Bit`, the input width must equal the number of bits per symbol.

The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

Dialog Box



M-ary number
The size of the alphabet.

Input type
Indicates whether the input consists of integers or groups of bits.

Symbol set ordering
Determines how the block maps each group of input bits to a corresponding integer. This field is active only when **Input type** is set to `Bit`.

Modulation index

Specify the modulation index $\{h_i\}$. The default is `0.5`. The value of this property must be a real, nonnegative scalar or column vector.

This block supports multi-h **Modulation index**. See CPM Modulator Baseband for details.

Phase offset (rad)

The initial phase of the output waveform, measured in radians.

Samples per symbol

The number of output samples that the block produces for each integer or binary word in the input, which must be a positive integer. For all non-binary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see "Upsample Signals and Rate Changes" in *Communications System ToolboxUser's Guide*.

Rate options

Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

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 *Communications System Toolbox Release Notes* for more information.

Output data type
Select the data type of the output signal. The output data type can be `single` or `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Boolean (When **Input type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (When **Input type** set to `Integer`)
Output	• Double-precision floating point • Single-precision floating point

Pair Block

CPFSK Demodulator Baseband

See Also

CPM Modulator Baseband, M-FSK Modulator Baseband

References

[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg. *Digital Phase Modulation*. New York: Plenum Press, 1986.

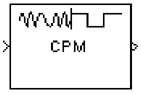
Purpose

Demodulate CPM-modulated data

Library

CPM, in Digital Baseband sublibrary of Modulation

Description



The Continuous Phase Modulation (CPM) Demodulator Baseband block demodulates a signal that was modulated using continuous phase modulation. The input is a baseband representation of the modulated signal:

$$s(t) = \exp\left[j2\pi \sum_{i=0}^{n} \alpha_i h_i q(t - iT) \right],$$
$$nT < t < (n+1)T$$

See the CPM Modulator Baseband block reference page for the definition of $\{\alpha_i\}$, $\{h_i\}$, and q(t).

This block accepts a scalar-valued or a column vector input signal with a data type of single or double. CPM is a modulation method with memory. The optimum receiver consists of a correlator followed by a maximum-likelihood sequence detector (MLSD) that searches the paths through the state trellis for the minimum Euclidean distance path. When the Modulation index *h* is rational, i.e., $h = m/p$, there are a finite number of phase states and the block uses the Viterbi algorithm to perform MLSD.

$\{h_i\}$ represents a sequence of modulation indices that moves cyclically through a set of indices $\{h_0, h_1, h_2, \ldots, h_{H-1}\}$.

- $h_i = m_i/p_i$ represents the modulation index in proper rational form
- m_i represents the numerator of modulation index
- p_i represents the denominator of modulation index
- m_i and p_i are relatively prime positive numbers
- The Least Common Multiple (LCM) of $\{p_0, p_1, p_2, \ldots, p_{H-1}\}$ is denoted as *p*

- $h_i = m'_i/p$

$\{h_i\}$ determines the number of phase states:

$$numPhaseStates = \begin{Bmatrix} p, \text{ for all even } m'_i \\ 2p, \text{ for any odd } m'_i \end{Bmatrix}$$

and affects the number of trellis states:

$numStates = numPhaseStates*M^{(L\text{-}1)}$

where

- *L* represents the **Pulse length**
- *M* represents the **M-ary number**

Integer-Valued Signals and Binary-Valued Signals

When you set the **Output type** parameter to `Integer`, then the block produces odd integers between -(M-1) and M-1. When you set the **Output type** to `Integer`, you cannot set **Output datatype** to `boolean`.

When you set the **Output type** parameter to `Bit`, then the block produces groupings of *K* bits. Each grouping is called a binary *word*. When you set the **Output type** to `Bit`, the **Output datatype** can only be `double` or `boolean`.

In binary output mode, the block first maps each input symbol to an intermediate value as in the integer output mode. Then, the block maps the odd integer k to the nonnegative integer (k+M-1)/2. Finally, the block maps each nonnegative integer to a binary word, using a mapping that depends on whether you set the **Symbol set ordering** parameter to `Binary` or `Gray`.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input width must be an integer multiple of the **Samples per symbol** parameter value, and the input can be a column vector.

- When you set **Output type** to `Bit`, the output width is K times the number of input symbols.
- When you set **Output type** to `Integer`, the output width is the number of input symbols.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is the product of the input sample time and the **Samples per symbol** parameter value.

- When you set **Output type** to `Bit`, the output width equals the number of bits per symbol.
- When you set **Output type** to `Integer`, the output is a scalar.

Traceback Depth and Output Delays

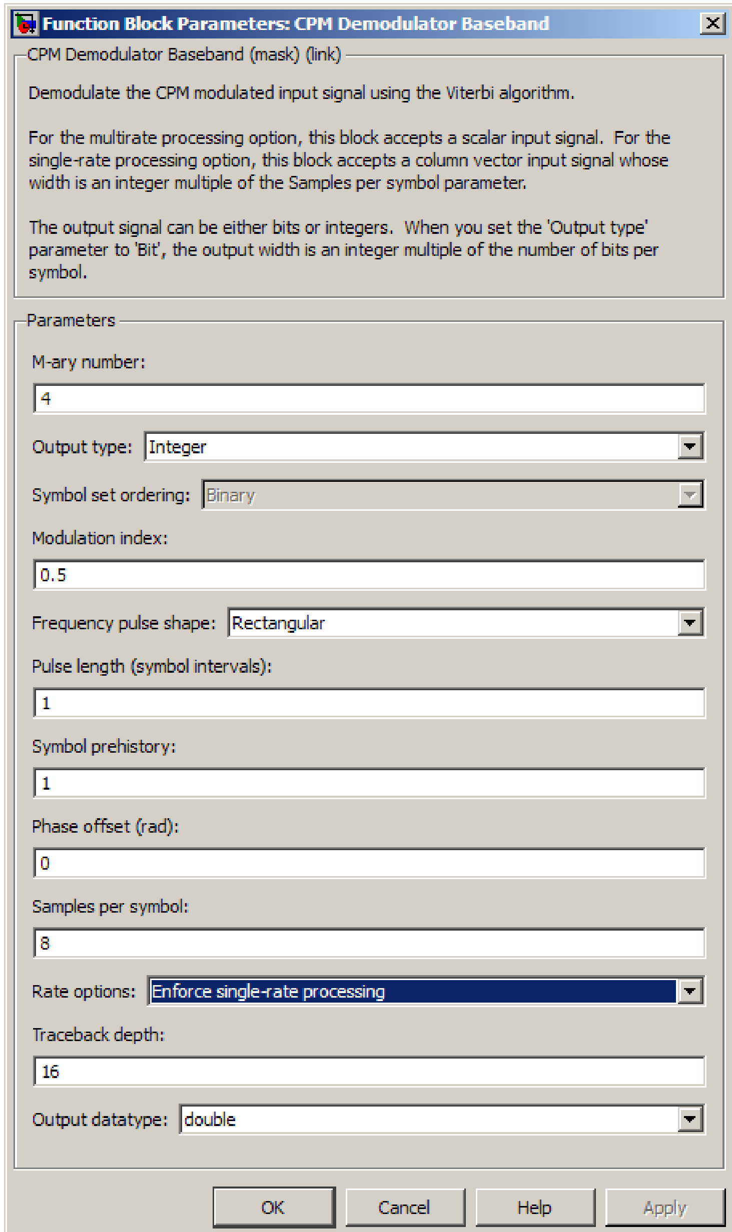
The **Traceback depth** parameter, D, in this block is the number of trellis branches used to construct each traceback path. D influences the output delay, which is the number of zero symbols that precede the first meaningful demodulated value in the output.

- When you set the **Rate options** parameter to `Allow multirate processing`, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking`, then the delay consists of D+1 zero symbols.
- When you set the **Rate options** parameter to `Enforce single-rate processing`, the delay consists of D zero symbols.

The optimal **Traceback depth** parameter value is dependent on minimum squared Euclidean distance calculations. Alternatively, a typical value, dependent on the number of states, can be chosen using the “five-times-the-constraint-length” rule, which corresponds to $5 \cdot \log 2(numStates)$.

For a binary raised cosine pulse shape with a pulse length of 3, h=2/3, this rule $(5 * \log 2(3 * 2^2) = 18)$ gives a result close to the optimum value of 20.

Dialog Box



M-ary number

The size of the alphabet.

Output type

Determines whether the output consists of integers or groups of bits.

Symbol set ordering

Determines how the block maps each integer to a group of output bits. This field applies only when you set **Output type** to `Bit`.

Modulation index

Specify the modulation index $\{h_i\}$. The default is `0.5`. The value of this property must be a real, nonnegative scalar or column vector.

Frequency pulse shape

Specify the type of pulse shaping that the corresponding modulator uses to smooth the phase transitions of the modulated signal. You can select from the following pulse shapes:

- `Rectangular`
- `Raised Cosine`
- `Spectral Raised Cosine`

  This option requires an additional parameter, **Rolloff**. The **Rolloff** parameter, which affects the spectrum of the pulse, is a scalar between zero and one.
- `Gaussian`

  This option requires an additional parameter, **BT product**. The **BT product** parameter, which represents bandwidth multiplied by time, is a nonnegative scalar. It is used to reduce the bandwidth at the expense of increased intersymbol interference.
- `Tamed FM` (tamed frequency modulation)

Main lobe pulse duration (symbol intervals)

Number of symbol intervals of the largest lobe of the spectral raised cosine pulse. This field is active only when **Frequency pulse shape** is set to `Spectral Raised Cosine`.

Rolloff
The rolloff factor of the specified raised cosine filter. This field appears only when you set **Frequency pulse shape** to `Spectral Raised Cosine`.

BT product
The product of bandwidth and time. This field appears only when **Frequency pulse shape** is set to `Gaussian`.

Pulse length (symbol intervals)
The length of the frequency pulse shape.

Symbol prehistory
The data symbols the modulator uses before the start of the simulation.

Phase offset (rad)
The initial phase of the modulated waveform.

Samples per symbol
The number of input samples that represent each modulated symbol. For more information, see "Upsample Signals and Rate Changes" in *Communications System ToolboxUser's Guide*.

Rate options
Select the rate processing method for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width is the number of symbols (which is given by dividing the input length by the **Samples per symbol** parameter value when the **Output type** parameter is set to `Integer`).
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output period is the same as the symbol period and equals the product of the input period and the **Samples per symbol** parameter value.

> **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 *Communications System Toolbox Release Notes* for more information.

For more information, see Single-Rate Processing and Multirate Processing in the Description section of this page.

Traceback depth
The number of trellis branches that the CPM Demodulator block uses to construct each traceback path.

Output datatype
The output data type can be `boolean`, `int8`, `int16`, `int32`, or `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Boolean (when **Output type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (when **Output type** set to `Integer`)

Pair Block

CPM Modulator Baseband

See Also

CPFSK Demodulator Baseband, GMSK Demodulator Baseband, MSK Demodulator Baseband, Viterbi Decoder

References

[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg. *Digital Phase Modulation*. New York: Plenum Press, 1986.

CPM Modulator Baseband

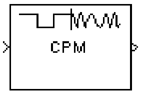
Purpose

Modulate using continuous phase modulation

Library

CPM, in Digital Baseband sublibrary of Modulation

Description



The Continuous Phase Modulation (CPM) Modulator Baseband block modulates an input signal using continuous phase modulation. The output is a baseband representation of the modulated signal:

$$s(t)=\exp\left[j2\pi\sum_{i=0}^{n}\alpha_i h_i q(t-iT)\right],$$
$$nT<t<(n+1)T$$

where

- $\{\alpha_i\}$ represents a sequence of M-ary data symbols selected from the alphabet ±1, ±3, ±(M -1).
- M must have the form 2^K for some positive integer K. You specify the value of M using the **M-ary number** parameter.
- $\{h_i\}$ represents a sequence of modulation indices and h_i moves cyclically through a set of indices $\{ h_0, h_1, h_2, \ldots, h_{H-1} \}$. When H =1, there is only one modulation index, h_0, which is denoted as h.

When h_i varies from interval to interval, the block operates in multi-h. To ensure a finite number of phase states, h_i must be a rational number. You specify the value(s) of h_i using the **Modulation index** parameter.

Continuous phase modulation uses pulse shaping to smooth the phase transitions of the modulated signal. The function q(t) is the phase response obtained from the frequency pulse, g(t), through the relation:

$$q(t)=\int_{-\infty}^{t} g(t)dt$$

Using the **Frequency pulse shape** parameter, you can select the following pulse shapes:

- `Rectangular`
- `Raised Cosine`
- `Spectral Raised Cosine`
- `Gaussian`
- `Tamed FM` (tamed frequency modulation)

For the exact definitions of these pulse shapes, see the work by Anderson, Aulin, and Sundberg among the references at the end of this page. Each pulse shape has a corresponding pulse duration. The **Pulse length (symbol intervals)** parameter measures this quantity in symbol intervals.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, then the block accepts odd integers between -(M-1) and M-1. *M* represents the **M-ary number** block parameter.

When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

where

K represents the number of bits per symbol.

The input vector length must be an integer multiple of *K*. In this configuration, the block accepts a group of *K* bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol, oversampled by the **Samples per symbol** parameter value, for each group of *K* bits.

This block accepts a scalar-valued or column vector input signal. For a column vector input signal, the width of the output frame equals the product of the number of symbols and the value for the **Samples per**

symbol parameter. For a sample-based input signal, the output sample time equals the symbol period divided by the value for the **Samples per symbol** parameter. For information about the data types each block port supports, see the "Supported Data Types" on page 2-160 table on this page.

Symbol Sets

In binary input mode, the block maps each binary word to an integer between 0 and M-1, using a mapping that depends on whether the **Symbol set ordering** parameter is set to `Binary` or `Gray`. The block then maps the integer k to the intermediate value 2k-(M-1) and proceeds as in the integer input mode. For more information, see Integer-Valued Signals and Binary-Valued Signals on the M-PSK Modulator ref page.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. In this mode, the input to the block can be multiple symbols.

- When you set **Input type** to `Integer`, the input can be a column vector, the length of which is the number of input symbols.
- When you set **Input type** to `Bit`, the input width must be an integer multiple of K, the number of bits per symbol.

The output width equals the product of the number of input symbols and the **Samples per symbol** parameter value.

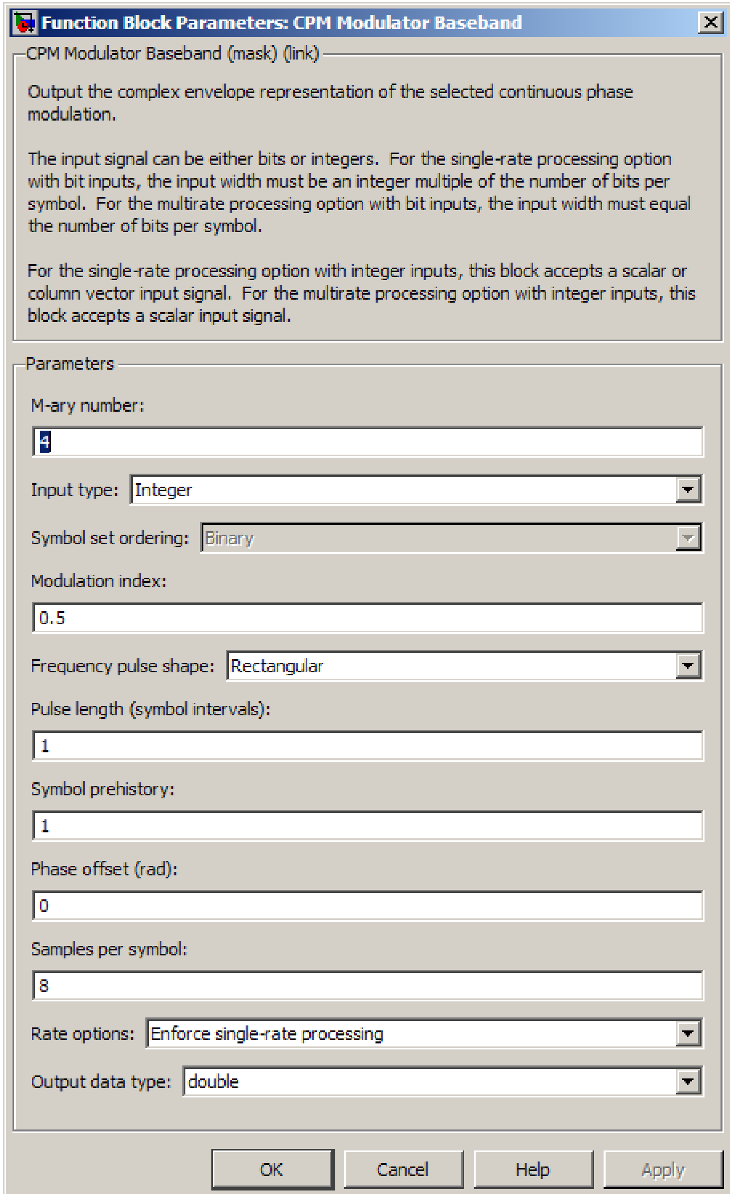
Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

- When you set **Input type** to `Integer`, the input must be a scalar.
- When you set **Input type** to `Bit`, the input width must equal the number of bits per symbol.

The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

Dialog Box



M-ary number

The size of the alphabet.

Input type
Indicates whether the input consists of integers or groups of bits.

Symbol set ordering
Determines how the block maps each group of input bits to a corresponding integer. This field is active only when **Input type** is set to `Bit`.

Modulation index
Specify the modulation index $\{h_i\}$. The default is `0.5`. The value of this property must be a real, nonnegative scalar or column vector.

Frequency pulse shape
Specify the type of pulse shaping that the block uses to smooth the phase transitions of the modulated signal. You can select from the following pulse shapes:

- `Rectangular`
- `Raised Cosine`
- `Spectral Raised Cosine`

  This option requires an additional parameter, **Rolloff**. The **Rolloff** parameter, which affects the spectrum of the pulse, is a scalar between zero and one.

- `Gaussian`

  This option requires an additional parameter, **BT product**. The **BT product** parameter, which represents bandwidth multiplied by time, is a nonnegative scalar. It is used to reduce the bandwidth at the expense of increased intersymbol interference.

- `Tamed FM` (tamed frequency modulation)

Main lobe pulse duration (symbol intervals)
Number of symbol intervals of the largest lobe of the spectral raised cosine pulse. This field is active only when **Frequency pulse shape** is set to `Spectral Raised Cosine`.

Rolloff

The rolloff factor of the specified spectral raised cosine pulse shape. This field appears only when you set **Frequency pulse shape** to `Spectral Raised Cosine`.

BT product

The product of bandwidth and time. This field appears only when **Frequency pulse shape** is set to `Gaussian`.

Pulse length (symbol intervals)

The length of the frequency pulse shape.

Symbol prehistory

A scalar-valued or vector signal that specifies the data symbols used before the start of the simulation, in reverse chronological order. If **Symbol prehistory** is a vector, then its length must be one less than the **Pulse length** parameter.

Phase offset (rad)

The initial phase of the output waveform, measured in radians.

Samples per symbol

The number of output samples that the block produces for each integer or binary word in the input, which must be a positive integer. For all non-binary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see "Upsample Signals and Rate Changes" in *Communications System ToolboxUser's Guide*.

Rate options

Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.

- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

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 *Communications System Toolbox Release Notes* for more information.

Output data type
Specify the block output data type as `double` and `single`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Boolean (when **Input type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (when **Input type** set to `Integer`)
Output	• Double-precision floating point • Single-precision floating point

Pair Block

CPM Demodulator Baseband

See Also

CPFSK Modulator Baseband, GMSK Modulator Baseband, MSK Modulator Baseband

References

[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg. *Digital Phase Modulation*. New York: Plenum Press, 1986.

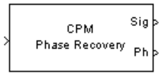
Purpose

Recover carrier phase using 2P-Power method

Library

Carrier Phase Recovery sublibrary of Synchronization

Description



The CPM Phase Recovery block recovers the carrier phase of the input signal using the 2P-Power method. This feedforward, non-data-aided, clock-aided method is suitable for systems that use these types of baseband modulation: continuous phase modulation (CPM), minimum shift keying (MSK), continuous phase frequency shift keying (CPFSK), and Gaussian minimum shift keying (GMSK). This block is suitable for use with blocks in the Baseband Continuous Phase Modulation library.

If you express the modulation index for CPM as a proper fraction, h = K / P, then P is the number to which the name "2P-Power" refers. The observation interval parameter must be an integer multiple of the input signal vector length.

The 2P-Power method assumes that the carrier phase is constant over a series of consecutive symbols, and returns an estimate of the carrier phase for the series. The **Observation interval** parameter is the number of symbols for which the carrier phase is assumed constant. This number must be an integer multiple of the input signal's vector length.

Input and Output Signals

This block accepts a scalar or column vector input signal of type `double` or `single`. The input signal represents a baseband signal at the symbol rate, so it must be complex-valued and must contain one sample per symbol.

The outputs are as follows:

- The output port labeled `Sig` gives the result of rotating the input signal counterclockwise, where the amount of rotation equals the carrier phase estimate. The `Sig` output is thus a corrected version of the input signal, and has the same sample time and vector size as the input signal.

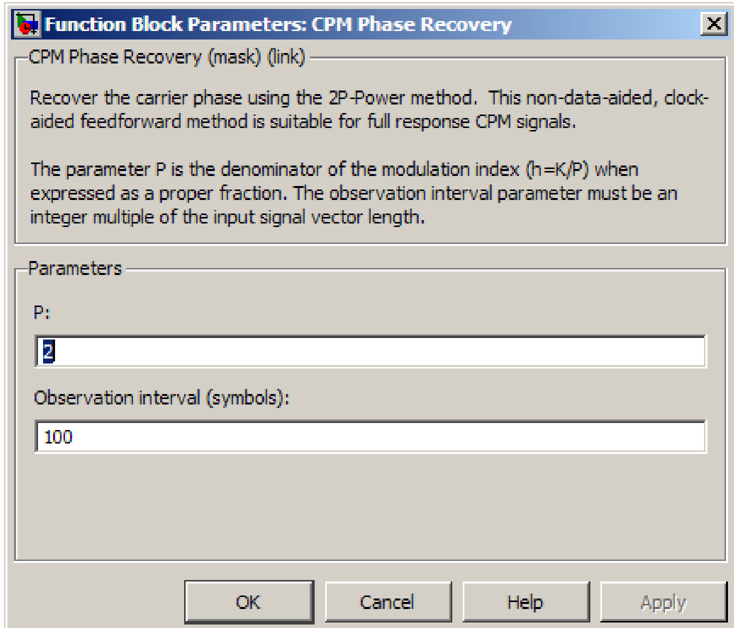
- The output port labeled `Ph` outputs the carrier phase estimate, in degrees, for all symbols in the observation interval. The `Ph` output is a scalar signal.

Note Because the block internally computes the argument of a complex number, the carrier phase estimate has an inherent ambiguity. The carrier phase estimate is between -90/P and 90/P degrees and might differ from the actual carrier phase by an integer multiple of 180/P degrees.

Delays and Latency

The block's algorithm requires it to collect symbols during a period of length **Observation interval** before computing a single estimate of the carrier phase. Therefore, each estimate is delayed by **Observation interval** symbols and the corrected signal has a latency of **Observation interval** symbols, relative to the input signal.

Dialog Box



P

The denominator of the modulation index for CPM (h = K / P) when expressed as a proper fraction.

Observation interval

The number of symbols for which the carrier phase is assumed constant. The observation interval parameter must be an integer multiple of the input signal vector length.

When this parameter is exactly equal to the vector length of the input signal, then the block always works. When the integer multiple is not equal to `1`, select **Simulation > Configuration Parameters > Solver**

and set **Tasking mode for periodic sample times** to `SingleTasking`.

Algorithm

If the symbols occurring during the observation interval are x(1), x(2), x(3),..., x(L), then the resulting carrier phase estimate is

$$\frac{1}{2P}\arg\left\{\sum_{k=1}^{L}(x(k))^{2P}\right\}$$

where the arg function returns values between -180 degrees and 180 degrees.

References

[1] Mengali, Umberto, and Aldo N. D'Andrea, *Synchronization Techniques for Digital Receivers*, New York, Plenum Press, 1997.

See Also

M-PSK Phase Recovery, CPM Modulator Baseband

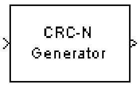
Purpose

Generate CRC bits according to CRC method and append to input data frames

Library

CRC sublibrary of Error Detection and Correction

Description



The CRC-N Generator block generates cyclic redundancy code (CRC) bits for each input data frame and appends them to the frame. The input must be a binary column vector. The CRC-N Generator block is a simplified version of the General CRC Generator block. With the CRC-N Generator block, you can select the generator polynomial for the CRC algorithm from a list of commonly used polynomials, given in the **CRC-N method** field in the block's dialog. N is degree of the generator polynomial. The table below lists the options for the generator polynomial.

CRC Method	Generator Polynomial	Number of Bits
CRC-32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1$	32
CRC-24	$x^{24}+x^{23}+x^{14}+x^{12}+x^{8}+1$	24
CRC-16	$x^{16}+x^{15}+x^{2}+1$	16
Reversed CRC-16	$x^{16}+x^{14}+x+1$	16
CRC-8	$x^{8}+x^{7}+x^{6}+x^{4}+x^{2}+1$	8
CRC-4	$x^{4}+x^{3}+x^{2}+x+1$	4

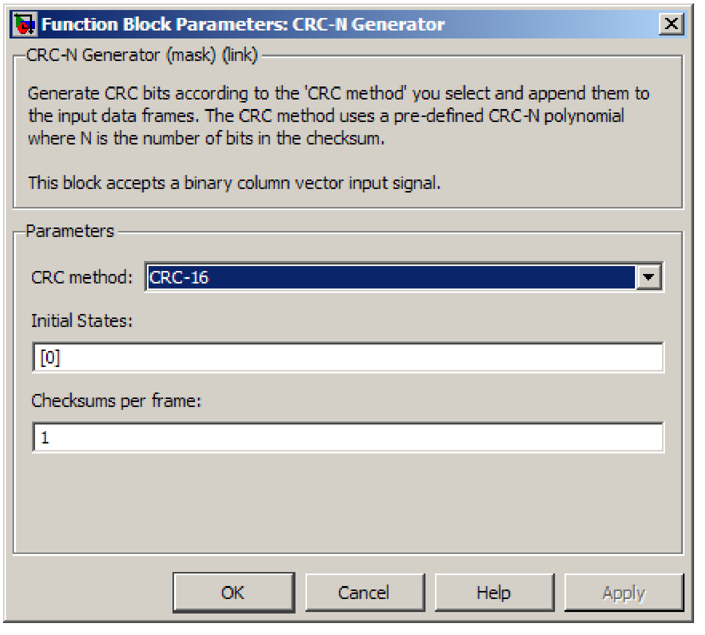
You specify the initial state of the internal shift register using the **Initial states** parameter. You specify the number of checksums that the block calculates for each input frame using the **Checksums per frame** parameter. For more detailed information, see the reference page for the General CRC Generator block.

This block supports `double` and `boolean` data types. The output data type is inherited from the input.

Signal Attributes

The General CRC Generator block has one input port and one output port. Both ports accept binary column vector input signals.

Dialog Box



CRC-N method

The generator polynomial for the CRC algorithm.

Initial states

A binary scalar or a binary row vector of length equal to the degree of the generator polynomial, specifying the initial state of the internal shift register.

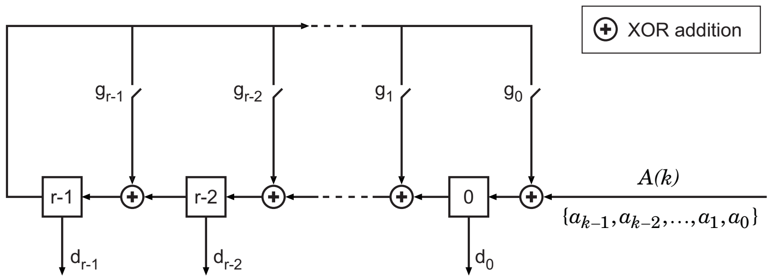
Checksums per frame

A positive integer specifying the number of checksums the block calculates for each input frame.

Algorithm

For a description of the CRC algorithm as implemented by this block, see "CRC Non-Direct Algorithm" in *Communications System Toolbox User's Guide*.

Schematic of the CRC Implementation



The above circuit divides the polynomial $a(x) = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0$ by $g(x) = g_{r-1}x^{r-1} + g_{r-2}x^{r-2} + \cdots + g_1 x + g_0$, and returns the remainder $d(x) = d_{r-1}x^{r-1} + d_{r-2}x^{r-2} + \cdots + d_1 x + d_0$.

The input symbols $\{a_{k-1}, a_{k-2}, \ldots, a_2, a_1, a_0\}$ are fed into the shift register one at a time in order of decreasing index. When the last symbol (a_0) works its way out of the register (achieved by augmenting the message with r zeros), the register contains the coefficients of the remainder polynomial $d(x)$.

This remainder polynomial is the checksum that is appended to the original message, which is then transmitted.

References

[1] Sklar, Bernard, *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, N.J., Prentice-Hall, 1988.

[2] Wicker, Stephen B., *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, N.J., Prentice Hall, 1995.

Pair Block

CRC-N Syndrome Detector

See Also

General CRC Generator, General CRC Syndrome Detector

Purpose

Detect errors in input data frames according to selected CRC method

Library

CRC sublibrary of Error Detection and Correction

Description

The CRC-N Syndrome Detector block computes checksums for its entire input frame. This block has two output ports. The first output port contains the set of message words with the CRC bits removed. The second output port contains the checksum result, which is a vector of a size equal to the number of checksums. A value of `0` indicates no checksum errors. A value of `1` indicates a checksum error occurred.

The CRC-N Syndrome Detector block is a simplified version of the General CRC Syndrome Detector block. You can select the generator polynomial for the CRC algorithm from a list of commonly used polynomials, given in the **CRC-N method** field in the block's dialog. N is the degree of the generator polynomial. The reference page for the CRC-N Generator block contains a list of the options for the generator polynomial.

The parameter settings for the CRC-N Syndrome Detector block should match those of the CRC-N Generator block.

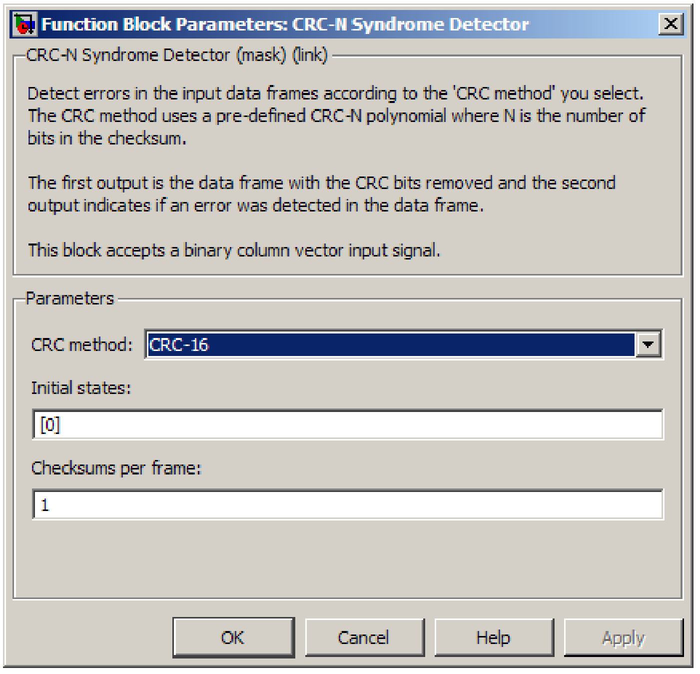
You specify the initial state of the internal shift register by the **Initial states** parameter. You specify the number of checksums that the block calculates for each input frame by the **Checksums per frame** parameter. For more detailed information, see the reference page for the General CRC Syndrome Detector block.

This block supports `double` and `boolean` data types. The output data type is inherited from the input.

Signal Attributes

The CRC-N Syndrome Detector block has one input port and two output ports. All three ports accept binary column vector signals.

Dialog Box



CRC-N method

The generator polynomial for the CRC algorithm.

Initial states

A binary scalar or a binary row vector of length equal to the degree of the generator polynomial, specifying the initial state of the internal shift register.

Checksums per frame

A positive integer specifying the number of checksums the block calculates for each input frame.

Algorithm

For a description of the CRC algorithm as implemented by this block, see “Cyclic Redundancy Check Codes” in *Communications System Toolbox User’s Guide*.

References

[1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, N.J., Prentice-Hall, 1988.

[2] Wicker, Stephen B., *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, N.J., Prentice Hall, 1995.

Pair Block

CRC-N Generator

See Also

General CRC Generator, General CRC Syndrome Detector

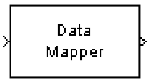
Purpose

Map integer symbols from one coding scheme to another

Library

Utility Blocks

Description



The Data Mapper block accepts integer inputs and produces integer outputs. You can select one of four mapping modes: `Binary to Gray`, `Gray to Binary`, `User Defined`, or `Straight Through`.

This block accepts a scalar, column vector, or full matrix input signal. It can accept multichannel inputs and allows for input and output data types of `double`, `single`, `int32`, `int16`, `int8`, `uint32`, `uint16`, and `uint8`. The input signal must be a non-negative value. The block truncates non-integer input signals as integer values.

Gray coding is an ordering of binary numbers such that all adjacent numbers differ by only one bit. However, the inputs and outputs of this block are integers, not binary vectors. As a result, the first two mapping modes perform code conversions as follows:

- In the `Binary to Gray` mode, the output from this block is the integer equivalent of the Gray code bit representation for the input integer.
- In the `Gray to Binary` mode, the output from this block is the integer position of the binary equivalent of the input integer in a Gray code ordering.

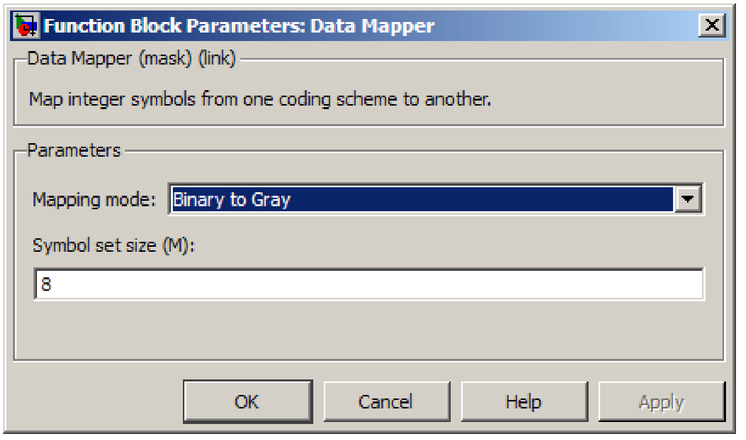
As an example, the table below shows both the `Binary to Gray` and `Gray to Binary` mappings for integers in the range 0 to 7. In the Binary to Gray Mode Output column, notice that binary representations in successive rows differ by exactly one bit. In the Gray to Binary Mode columns, notice that sorting the rows by Output value creates a Gray code ordering of Input binary representations.

Binary to Gray Mode		Gray to Binary Mode	
Input	**Output**	**Input**	**Output**
0	0 *(000)*	0 *(000)*	0
1	1 *(001)*	1 *(001)*	1
2	3 *(011)*	2 *(010)*	3
3	2 *(010)*	3 *(011)*	2
4	6 *(110)*	4 *(100)*	7
5	7 *(111)*	5 *(101)*	6
6	5 *(101)*	6 *(110)*	4
7	4 *(100)*	7 *(111)*	5

When you select the `User Defined` mode, you can use any arbitrary mapping by providing a vector to specify the output ordering. For example, the vector `[1,5,0,4,2,3]` defines the following mapping:

$$0 \to 1$$
$$1 \to 5$$
$$2 \to 0$$
$$3 \to 4$$
$$4 \to 2$$
$$5 \to 3$$

When you select the `Straight Through` mode, the output equals the input.



Dialog Box

Mapping mode
: The type of data mapping that the block performs.

Symbol set size
: Symbol set size of M restricts this block's inputs and outputs to integers in the range 0 to M-1.

Mapping vector
: A vector of length M that contains the integers from 0 to M-1. The order of the elements of this vector specifies the mapping of inputs to outputs. This parameter appears only when you set **Mapping mode** to `User Defined`.

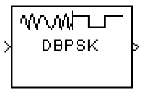
Purpose

Demodulate DBPSK-modulated data

Library

PM, in Digital Baseband sublibrary of Modulation

Description

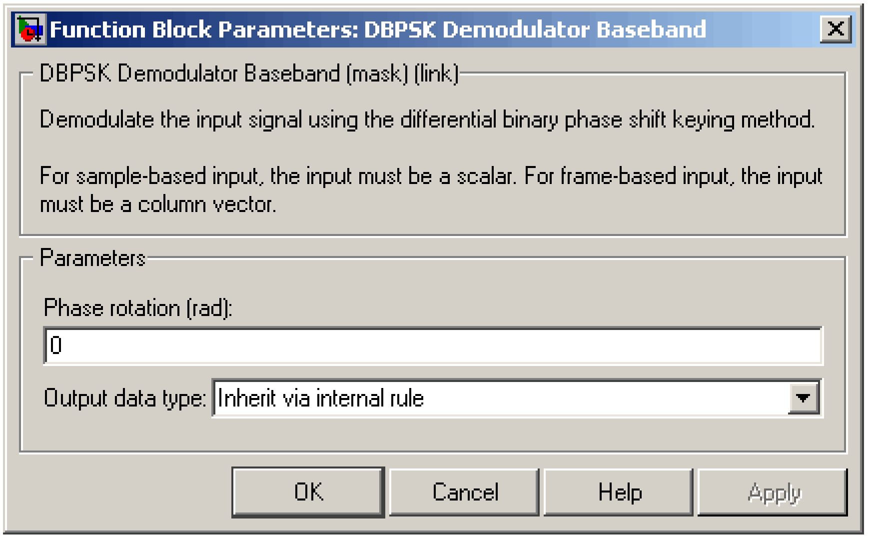


The DBPSK Demodulator Baseband block demodulates a signal that was modulated using the differential binary phase shift keying method. The input is a baseband representation of the modulated signal.

The input must be a discrete-time complex signal. The block compares the current symbol to the previous symbol. It maps phase differences of θ and π+θ, respectively, to outputs of 0 and 1, respectively, where θ is the **Phase rotation** parameter. The first element of the block's output is the initial condition of zero because there is no previous symbol with which to compare the first symbol.

This block accepts a scalar or column vector input signal. The input signal can be of data types `single` and `double`. For information about the data types each block port supports, see "Supported Data Types" on page 2-177.

Dialog Box



Phase rotation (rad)
This phase difference between the current and previous modulated symbols results in an output of zero.

Output data type
When the parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is of type `single` or `double`.

For additional information, see "Supported Data Types" on page 2-177.

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 • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

DBPSK Modulator Baseband

See Also

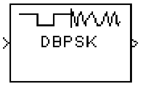
M-DPSK Demodulator Baseband, DQPSK Demodulator Baseband, BPSK Demodulator Baseband

DBPSK Modulator Baseband

Purpose Modulate using differential binary phase shift keying method

Library PM, in Digital Baseband sublibrary of Modulation

Description The DBPSK Modulator Baseband block modulates using the differential binary phase shift keying method. The output is a baseband representation of the modulated signal.

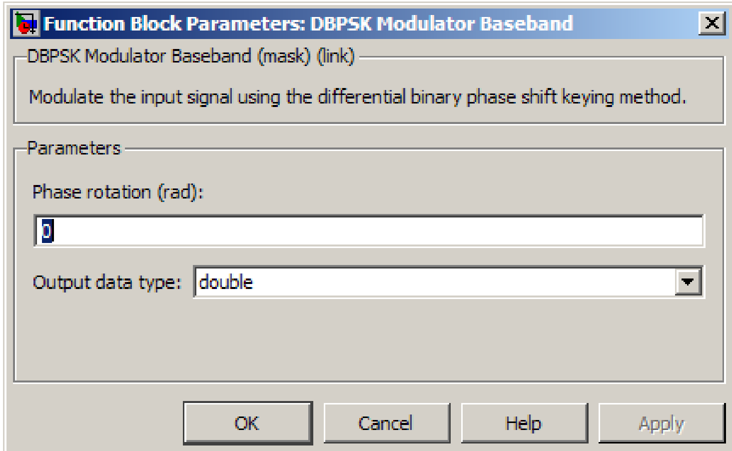


This block accepts a scalar or column vector input signal. The input must be a discrete-time binary-valued signal. For information about the data types each block port supports, see “Supported Data Types” on page 2-179.

The following rules govern this modulation method when the **Phase rotation** parameter is θ:

- If the first input bit is 0 or 1, respectively, then the first modulated symbol is exp(jθ) or -exp(jθ), respectively.
- If a successive input bit is 0 or 1, respectively, then the modulated symbol is the previous modulated symbol multiplied by exp(jθ) or -exp(jθ), respectively.

Dialog Box



Phase rotation (rad)
: The phase difference between the previous and current modulated symbols when the input is zero.

Output Data type
: The output data type can be either `single` or `double`. By default, the block sets this to `double`.

Supported Data Types

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

Pair Block DBPSK Demodulator Baseband

See Also DQPSK Modulator Baseband, BPSK Modulator Baseband

Purpose

Distribute elements of input vector alternately between two output vectors

Library

Sequence Operations

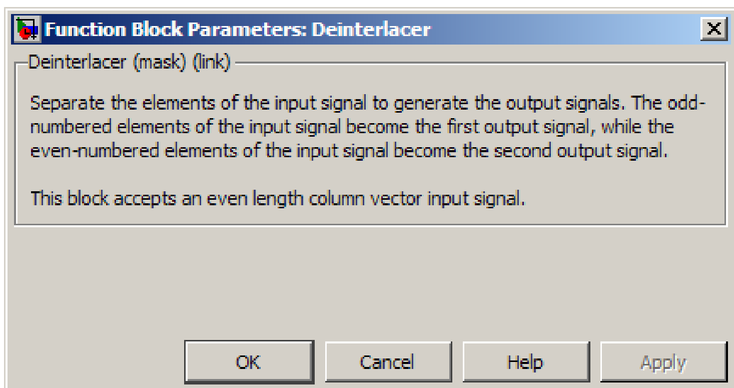
Description

The Deinterlacer block accepts an even length column vector input signal. The block alternately places the elements in each of two output vectors. As a result, each output vector size is half the input vector size. The output vectors have the same complexity and sample time of the input.

This block accepts a column vector input signal with an even integer length. The block supports the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

The Deinterlacer block can be useful for separating in-phase and quadrature information from a single vector into separate vectors.

Dialog Box



Examples

If the input vector has the values [1; 5; 2; 6; 3; 7; 4; 8], then the two output vectors are [1; 2; 3; 4] and [5; 6; 7; 8]. Notice that this example is the inverse of the example on the reference page for the Interlacer block.

If the input vector has the values [1; 2; 3; 4; 5; 6], then the two output vectors are [1; 3; 5] and [2; 4; 6].

Pair Block

Interlacer

See Also

Demux (Simulink documentation)

Purpose

Reduce sampling rate by averaging consecutive samples

Library

Sequence Operations

Description

The Derepeat block resamples the discrete input at a rate 1/N times the input sample rate by averaging N consecutive samples. This is one possible inverse of the DSP System Toolbox Repeat block. The positive integer N is the **Derepeat factor** parameter in the Derepeat dialog.

The **Initial condition** parameter prescribes elements of the output when it is still too early for the input data to show up in the output. If the dimensions of the **Initial condition** parameter match the output dimensions, then the parameter represents the initial output value. If **Initial condition** is a scalar, then it represents the initial value of each element in the output. The block does not support empty matrices for initial conditions.

The input can have any shape or frame status. The block accepts the data types `single` and `double`. The output signal inherits its data type from the input signal.

This block works within a triggered subsystem, as long as you use it in the single-rate mode.

Single-Rate Processing

The block derepeats each frame, treating distinct channels independently. Each element of the output is the average of N consecutive elements along a *column* of the input matrix. The **Derepeat factor** must be less than the frame size.

When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. The block reduces the sampling rate by using a proportionally smaller frame *size* than the input. **Derepeat factor** should be an integer factor of the number of rows in the input vector or matrix. For derepetition by a factor of N, the output frame size is 1/N times the input frame size, but the input and output frame rates are equal. When you use this option, the **Initial condition** parameter does not apply and the block incurs no delay, because the input data immediately shows up in the output.

For example, if a single-channel input with 64 elements is derepeated by a factor of 4, then the output contains 16 elements. The input and output frame periods are equal.

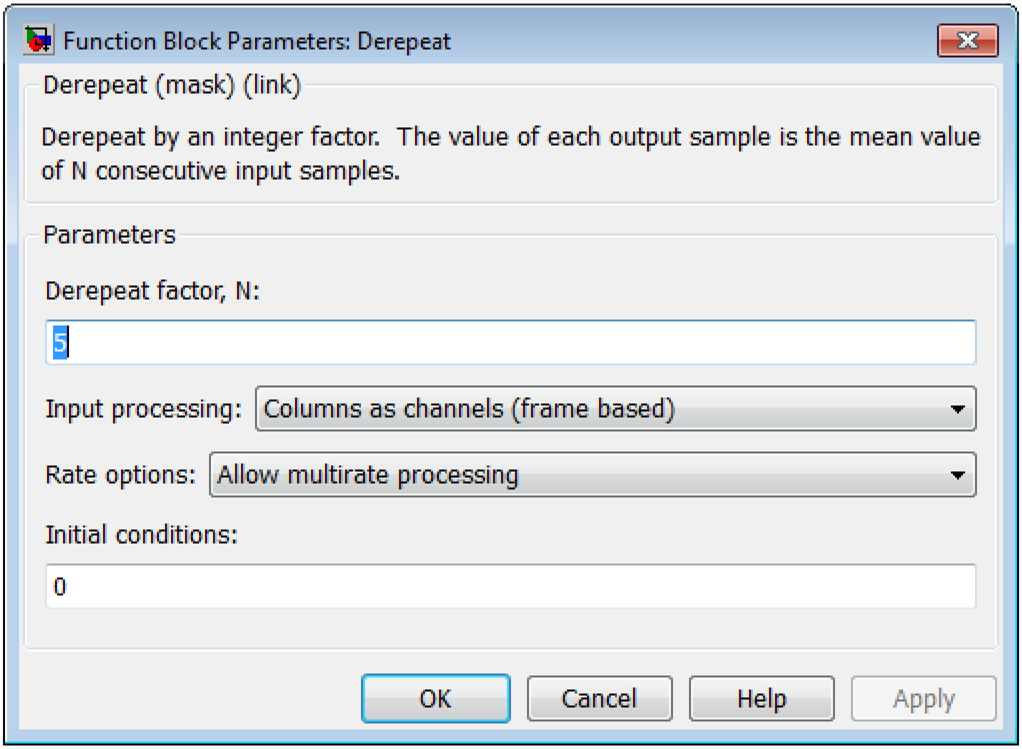
Multirate Processing

When you set the **Rate options** parameter to `Allow multirate processing`, the input and output of the Derepeat block are the same size, but the sample rate of the output is *N* times slower than that of the input. When the block is in multirate processing mode, you must also specify a value for the **Input processing** parameter:

- When you set the **Input processing** parameter to `Elements as channels (sample based)`, then the block assumes that the input is a vector or matrix whose elements represent samples from independent channels. The block averages samples from each channel independently over time. The output period is N times the input period, and the input and output sizes are identical. The output is delayed by one output period, and the first output value is the **Initial condition** value. If you set **Rate options** to `Enforce single-rate processing`, the block will generate an error message.
- When you set the **Input processing** parameter to `Columns as channels (frame based)`, The block reduces the sampling rate by using a proportionally longer frame *period* at the output port than at the input port. For derepetition by a factor of N, the output frame period is N times the input frame period, but the input and output frame sizes are equal. The output is delayed by one output frame, and the first output frame is determined only by the **Initial condition** value. The block derepeats each frame, treating distinct channels independently. Each element of the output is the average of N consecutive elements along a *column* of the input matrix. The **Derepeat factor** must be less than the frame size.

For example, if a single-channel input with a frame period of 1 second is derepeated by a factor of 4, then the output has a frame period of 4 seconds. The input and output frame sizes are equal.

Dialog Box



Derepeat factor, N

The number of consecutive input samples to average in order to produce each output sample.

Input processing

Specify how the block processes the input signal. 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 *Communications System Toolbox Release Notes* for more information.

Rate options

Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

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 *Communications System Toolbox Release Notes* for more information.

Initial condition
The value with which to initialize the block.

See Also

Repeat (DSP System Toolbox documentation), Downsample (DSP System Toolbox documentation)

Descrambler

Purpose

Descramble input signal

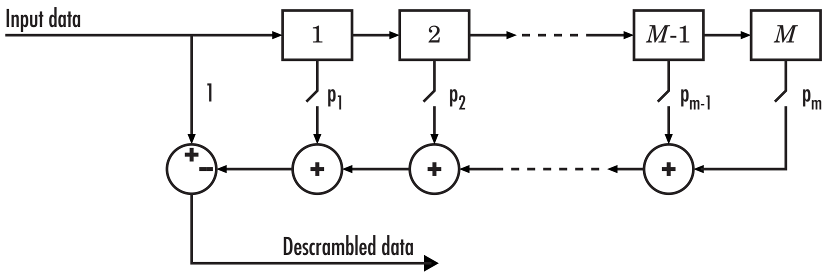
Library

Sequence Operations

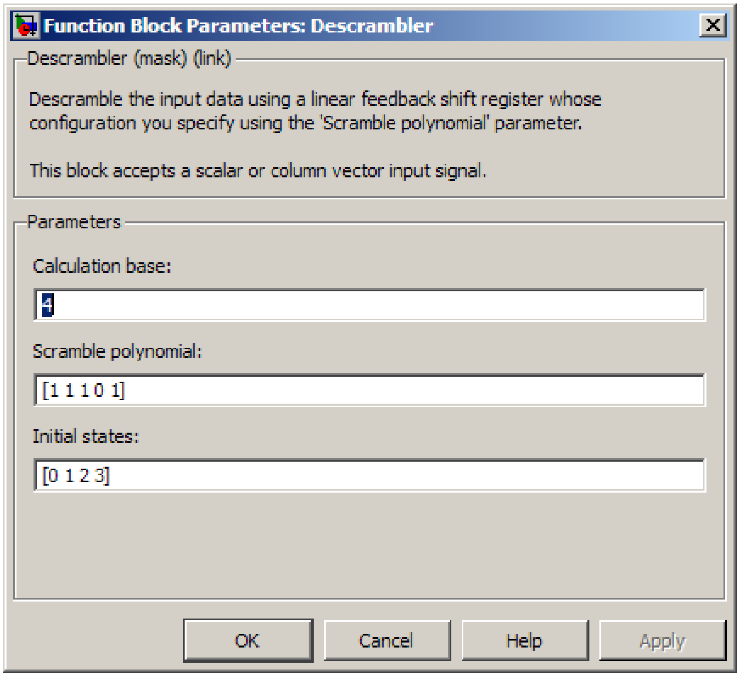
Description

The Descrambler block descrambles a scalar or column vector input signal. The Descrambler block is the inverse of the Scrambler block. If you use the Scrambler block in a transmitter, then you use the Descrambler block in the related receiver.

In the following descrambler schematic, the adders and subtracter operate modulo *N*, where *N* is the **Calculation base** parameter. You must specify integer input values between 0 and N-1.



At each time step, the input causes the contents of the registers to shift sequentially. Using the **Scramble polynomial** parameter, you specify if each switch in the descrambler is on or off. To make the Descrambler block reverse the operation of the Scrambler block, use the same **Scramble polynomial** parameters in both blocks. If there is no signal delay between the scrambler and the descrambler, then the **Initial states** in the two blocks must be the same. See the reference page for the Scrambler block for more information about these parameters.



Dialog Box

Calculation base

The calculation base N. The input and output of this block are integers in the range [0, N-1].

Scramble polynomial

A polynomial that defines the connections in the scrambler.

Initial states

The states of the scrambler's registers when the simulation starts.

Pair Block

Scrambler

Differential Decoder

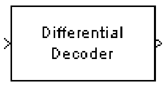
Purpose

Decode binary signal using differential coding

Library

Source Coding

Description



The Differential Decoder block decodes the binary input signal. The output is the logical difference between the consecutive input element within a channel. More specifically, the block's input and output are related by

$m(i_0) = d(i_0)$ XOR **Initial condition** parameter value

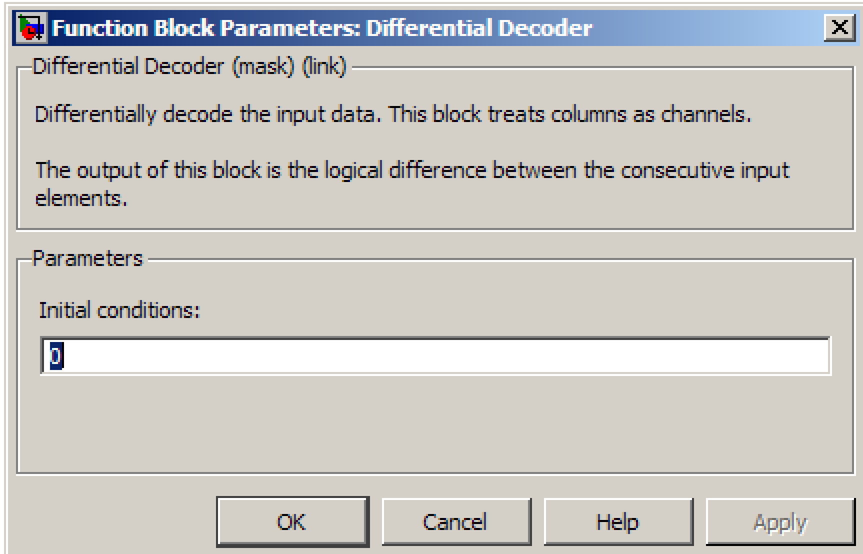
$m(i_k) = d(i_k)$ XOR $d(i_{k-1})$

where

- d is the differentially encoded input.
- m is the output message.
- i_k is the kth element.
- XOR is the logical exclusive-or operator.

This block accepts a scalar, column vector, or matrix input signal and treats columns as channels.

Dialog Box



Initial conditions

The logical exclusive-or of this value with the initial input value forms the initial output value.

Supported Data Type

Port	Supported Data Types
In	• double • single • boolean • integer • fixed-point
Out	• double • single • boolean

Port	Supported Data Types
	• integer • fixed-point

References

[1] Couch, Leon W., II, *Digital and Analog Communication Systems*, Sixth edition, Upper Saddle River, N. J.: Prentice Hall, 2001.

Pair Block

Differential Encoder

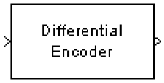
Purpose

Encode binary signal using differential coding

Library

Source Coding

Description



The Differential Encoder block encodes the binary input signal within a channel. The output is the logical difference between the current input element and the previous output element. More specifically, the input and output are related by

$d(i_0) = m(i_0)$ XOR **Initial condition** parameter value

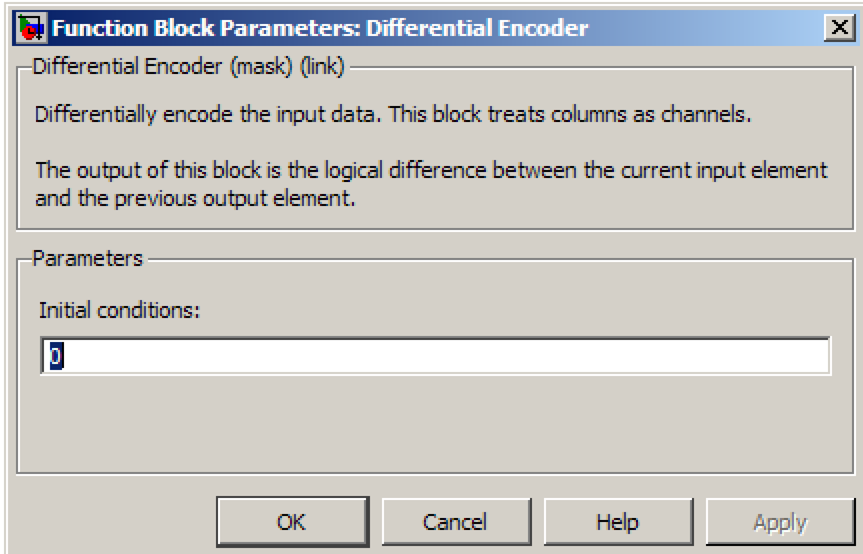
$d(i_k) = d(i_{k-1})$ XOR $m(i_k)$

where

- m is the input message.
- d is the differentially encoded output.
- i_k is the kth element.
- XOR is the logical exclusive-or operator.

This block accepts a scalar or column vector input signal and treats columns as channels.

Dialog Box



Initial conditions

The logical exclusive-or of this value with the initial input value forms the initial output value.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • Integer • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Boolean

Port	Supported Data Types
	• Integer • Fixed-Point

References

[1] Couch, Leon W., II, *Digital and Analog Communication Systems*, Sixth edition, Upper Saddle River, N. J., Prentice Hall, 2001.

Pair Block

Differential Decoder

Discrete-Time Eye Diagram Scope

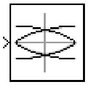
Purpose

Display multiple traces of modulated signal

Library

Comm Sinks

Description



The Discrete-Time Eye Diagram Scope block displays multiple traces of a modulated signal to produce an eye diagram. You can use the block to reveal the modulation characteristics of the signal, such as pulse shaping or channel distortions.

The Discrete-Time Eye Diagram Scope block has one input port. This block accepts a scalar-valued or column vector input signal. The block accepts a signal with the following data types: `double`, `single`, `boolean`, base integer, and fixed-point data types for input, but casts as `double` prior to displaying the results.

Marker and Line Styles

The **Marker**, **Line style**, and **Line color** parameters, on the **Rendering Properties** panel, control the appearance of the signal trajectory. The **Marker** parameter specifies the marker style for points in the eye diagram. The following table lists some of the available line markers.

Marker Style	Parameter Symbol	Appearance
Plus	`+`	
Circle	`o`	
Asterisk	`*`	
Point	`.`	
Cross	`x`	

The **Line style** parameter specifies the style for lines in the eye diagram. The following lists some of the available line styles.

Line Style	Appearance
Solid	
Dashed	
Dotted	
Dash-dot	

The **Line color** parameter specifies the color of the eye diagram. These settings plot the signal channels in the following colors (8-bit RGB equivalents are 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)	

See the `line` function in the MATLAB documentation for more information about the available markers, colors, and line styles.

Recommended Settings

The following table summarizes the recommended parameter settings for the Discrete-Time Eye Diagram Scope.

Parameter	Recommended Setting
Samples per symbol	Same as the **Samples per symbol** setting in the modulator block, or the **Interpolation factor** setting in the interpolation block
Offset (samples)	0 to view the open part of the eye (**Samples per symbol**)/2 to view the closed part of the eye
Symbols per trace	An integer between 1 and 4
Traces displayed	10 times the alphabet size of the modulator, M
New traces per display	Same as **Traces displayed** for greater speed A small positive integer for best animation
Marker	None or a point (.) to see where the samples are plotted
Line style	Solid dash (-)
Line color	Blue (b)
Duplicate points at trace boundary	Check **Duplicate points at trace boundary** for modulations such as PSK and QAM. Clear to display the phase trees for MSK, CPFSK, GFSK, GMSK, and other continuous phase modulations.
Color fading	Check **Color fading** for animation that resembles an oscilloscope. Clear for greater speed and animation that resembles a plot.

Parameter	Recommended Setting
High quality rendering	Check **High quality rendering** for better animation. Clear for greater speed.
Eye diagram to display	Select `In-phase and Quadrature` to view real and imaginary components. Select `In-phase Only` to view real component only and for greater speed. When the input is real and you choose `In-phase and Quadrature`, the quadrature component of the eye diagram is zero.
Open at start of simulation	Check **Open at start of simulation** to view the signal at the start of simulation. Clear to view the signal after convergence to steady state and for greater initial speed.
Y-axis minimum	Approximately 10% less than the expected minimum value of the signal
Y-axis maximum	Approximately 10% greater than the expected maximum value of the signal

For Rapid Accelerator or External mode, set the scope up for single rate mode. To guarantee the satisfactory behavior of single rate mode, the subsystem below the block mask for this block must operate as a single-rate entity, which means the following conditions are true:

sps * ((td * (spt-1)) + ntpd) = Sf

where:

- sps = Samples per symbol
- td = Traces displayed
- spt = Symbols per trace
- ntpd = New traces per display
- Sf = Input frame size, in samples

This equation guarantees that the subsystem below the mask for this block operates as a single rate entity.

Warning

If you want to use Rapid Accelerator or External mode, set this block up to run as a single rate entity because the block does not support multi-rate in these modes.

Note Before running a model that contains a Discrete-Time Eye Diagram Scope block in Accelerator, Rapid Accelerator, or External mode, you must select **Open scope at start of simulation**. If you do not select this check box before running your model for the first time, the scope will not display your simulation data

---

Scope Options

The scope title (in the window title bar) is the same as the block title. You can set the axis scaling by setting the y-axis minimum and y-axis maximum parameters on the **Axes Properties** panel.

In addition to the standard MATLAB figure window menus (**File**, **Edit**, **Window**, **Help**), the Vector Scope window has an **Axes** and a **Channels** menu.

The properties listed in the **Axes** menu apply to all channels. Many of the parameters in this menu are also accessible through the block parameter dialog box. These are **Autoscale**, **Show grid**, **Frame #**, and **Save Position**. Below are descriptions of the other parameters listed in the **Axes** menu:

- **Autoscale** resizes the *y*-axis to best fit the vertical range of the data. The numerical limits selected by the autoscale feature are displayed in the **Minimum Y-limit** and **Maximum Y-limit** parameters in the parameter dialog box. You can change them by editing those values.
- **Show grid** - When selected, the scope displays a grid according to tick marks on the *x*- and *y*-axes.
- **Frame #** - When selected, the scope displays the current frame number at the bottom of the scope window.
- **Save Position** automatically updates the **Scope position** parameter in the **Figure properties** panel to reflect the scope window's current position and size. To make the scope window open at a particular location on the screen when the simulation runs, simply drag the window to the desired location, resize it as needed, and select **Save Position**.

The properties listed in the **Channels** menu apply to a particular channel. The parameters listed in this menu are **Style**, **Marker**, and **Color**. They correspond to the parameters **Line style**, **Marker**, and **Line color**, respectively.

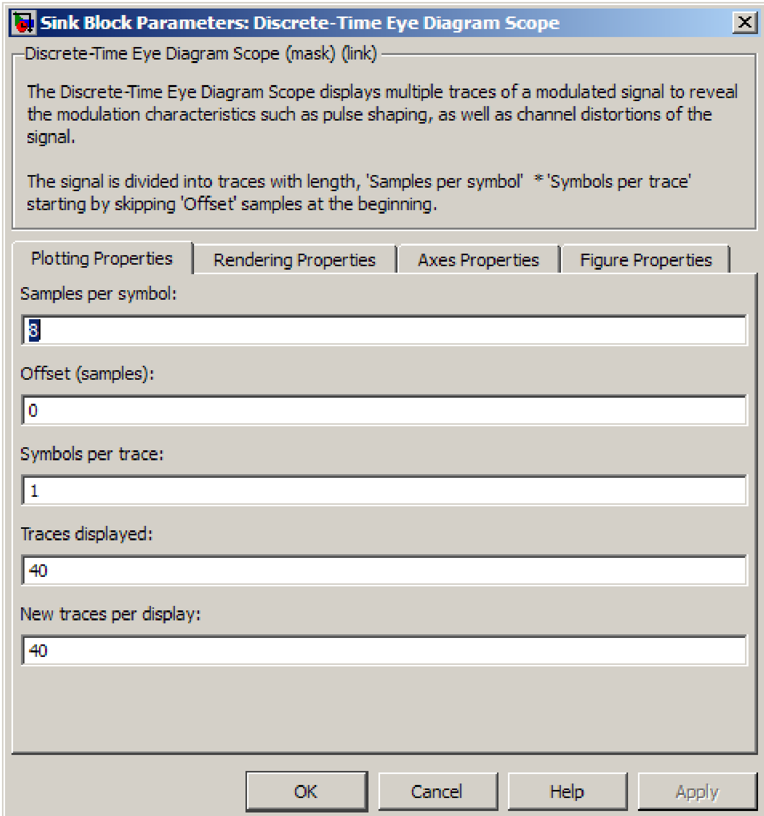
You can also access many of these options by right-clicking with the mouse anywhere on the scope display. The menu that pops up contains a combination of the options available in both the **Axes** and **Channels** menus.

Behavior in Enabled Subsystems

You can use the Discrete-Time Eye Diagram Scope block inside an enabled subsystem. However, you cannot use the scope block inside an enabled subsystem when the model is in a multirate multitasking environment.

When you use the scope in a multirate singletasking environment, it may generate unexpected results inside enabled subsystems. To workaround this issue, configure the scope for single-rate mode. See “Recommended Settings” on page 2-197 for the parameter settings that enable single-rate mode.

Dialog Box



Samples per symbol

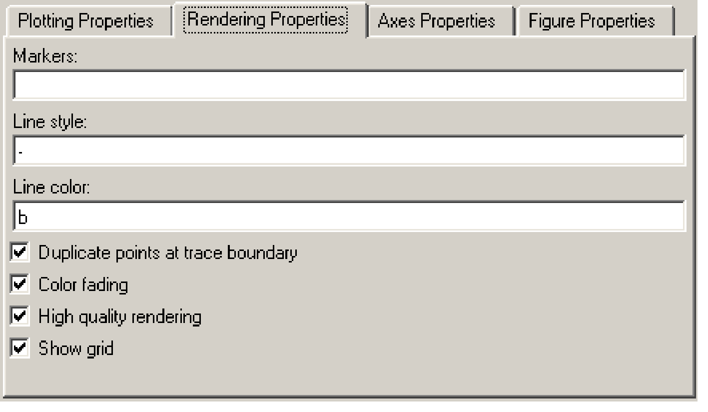
Number of samples per symbol. Use with **Symbols per trace** to determine the number of samples per trace.

Offset (samples)
: Nonnegative integer less than the product of **Samples per symbol** and **Symbols per trace**, specifying the number of samples to omit before plotting the first point. Tunable.

Symbols per trace
: Positive integer specifying the number of symbols plotted per trace.

Traces displayed
: Number of traces plotted.

New traces per display
: Positive integer less than **Traces displayed**, specifying the number of new traces that appear in each display.



Markers
: The marker for points in the eye diagram. Tunable.

Line style
: The line style in the eye diagram. Tunable.

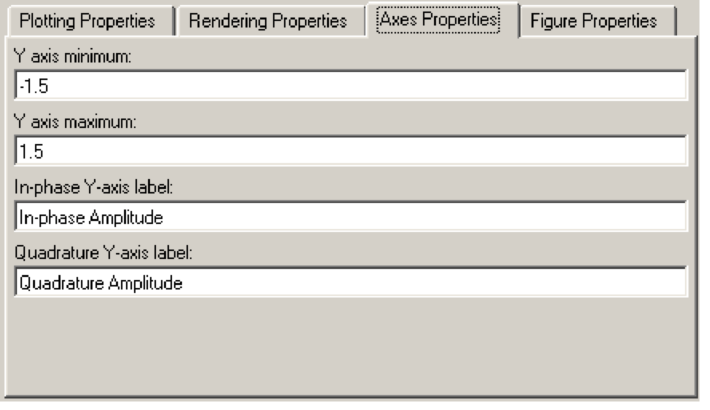
Line color
: The line color in the eye diagram. Tunable.

Duplicate points at trace boundary
: Check to enable duplicate points at the trace boundary. Clear to disable.

Color fading
: When selected, the points in the eye diagram fade as the interval of time after they are first plotted increases. Tunable.

High quality rendering
: When selected, the block renders a slow, higher-quality picture with overwrite raster operations. When cleared, the block renders a fast, lower-quality picture with XOR raster operations. Tunable.

Show grid
: Toggles the scope grid on and off. Tunable.



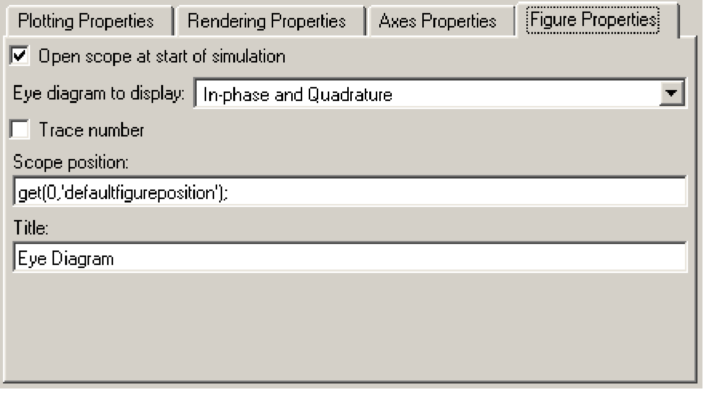
Y-axis minimum
: Minimum signal value the scope displays. Tunable.

Y-axis maximum
: Maximum signal value the scope displays. Tunable.

In-phase Y-axis label
: Label for *y*-axis of the in-phase diagram. Tunable.

Quadrature Y-axis label

Label for *y*-axis of the quadrature diagram. Tunable.



Open scope at start of simulation

When selected, the scope opens at the start of simulation. When cleared, you must double-click the block after the start of simulation to open the scope. Tunable.

Note Before running a model that contains a Discrete-Time Eye Diagram Scope block in Accelerator, Rapid Accelerator, or External mode, you must select **Open scope at start of simulation**. If you do not select this check box before running your model for the first time, the scope will not display your simulation data

---

Eye diagram to display

Type of eye diagram to display. Choose `In-phase and Quadrature` to display real and complex components, or `In-phase Only` to display only the real component. Tunable.

Trace number
Displays the number of the current trace in the input sequenced. Tunable.

Scope position
A four-element vector of the form [left bottom width height] specifying the position of the scope window. (0,0) is the lower left corner of the display. Tunable.

Title
Title of eye diagram figure window. Tunable.

Examples

For documentation examples that use this block, see "View a Sinusoid" and "View a Modulated Signal".

Also, the following Communications System Toolbox demos illustrate how to use the Discrete-Time Eye Diagram Scope block:

- CPM Phase Tree Example
- Filtered Offset QPSK vs. Filtered QPSK
- Rayleigh Fading Channel
- QPSK vs. MSK

See Also

Discrete-Time Scatter Plot Scope, Discrete-Time Signal Trajectory Scope

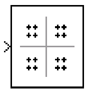
Purpose

Display in-phase and quadrature components of modulated signal constellation

Library

Comm Sinks

Description

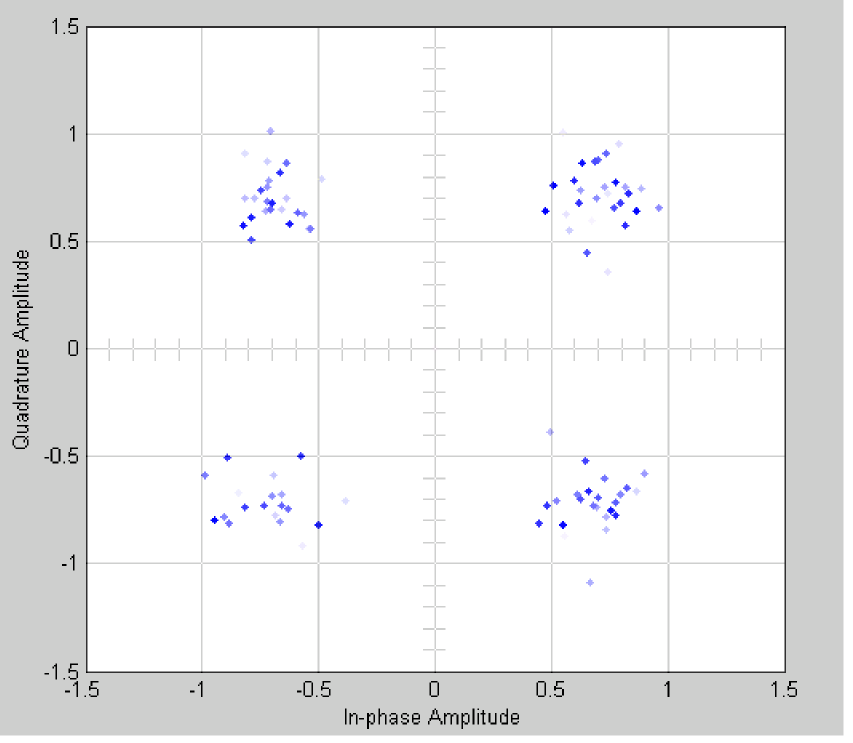


The Discrete-Time Scatter Plot Scope block displays scatter plots of a modulated signal, to reveal the modulation characteristics, such as pulse shaping or channel distortions of the signal.

The Discrete-Time Scatter Plot Scope block has one input port. This block accepts a complex scalar-valued or column vector input signal. The block accepts a signal with the following data types: `double`, `single`, base integer, and fixed-point for input, but will cast it as `double`.

See the reference page for the Discrete-Time Signal Trajectory Scope block to compare the preceding scatter plot with the trajectory of the same signal. The Discrete-Time Signal Trajectory Scope block connects the points displayed by the Discrete-Time Scatter Plot Scope block to display the signal trajectory.

Setting **Samples per symbol** to `8`, increasing **Points displayed** to `100`, and running the model for `100` seconds produces the following scatter plot.



Markers and Color

The **Markers** and **Color** parameters, on the **Rendering Properties** panel, specify the style and color of markers in the scatter plot. For details on the options for these parameters, see the reference page for the Discrete-Time Eye Diagram Scope block.

Recommended Settings

The following table summarizes the recommended parameter settings for the Discrete-Time Scatter Plot Scope.

Parameter	Recommended Setting
Samples per symbol	Same as the **Samples per symbol** setting in the modulator block, or the **Interpolation factor** setting in the interpolation block
Points displayed	10 times the alphabet size of the modulator
New points per display	Same as **Points displayed** for greater speed A small positive integer for best animation
Line style	Solid dash (-)
Line color	Blue (b)
Color fading	Check **Color fading** for animation that resembles an oscilloscope. Clear for greater speed and animation that resembles a plot.
High quality rendering	Check **High quality rendering** for higher quality rendering. Clear for greater speed.
Open at start of simulation	Check **Open at start of simulation** to view the signal at the start of simulation. Clear to view the signal after convergence to steady state and for greater initial speed.

Parameter	Recommended Setting
X-axis minimum	Approximately 10% less than the expected minimum value of the signal
X-axis maximum	Approximately 10% greater than the expected maximum value of the signal

For Rapid Accelerator or External mode, set the scope up for single rate mode. To guarantee the satisfactory behavior of single rate mode, the subsystem below the block mask for this block must operate as a single-rate entity, which means the following conditions are true:

$$sps * nppd = Sf$$

where:

- sps = Samples per symbol
- nppd = New points per display
- Sf = Input frame size, in samples

This equation guarantees that the subsystem below the mask for this block operates as a single rate entity.

Warning

If you want to use Rapid Accelerator or External mode, set this block up to run as a single rate entity because multi-rate does not support these modes.

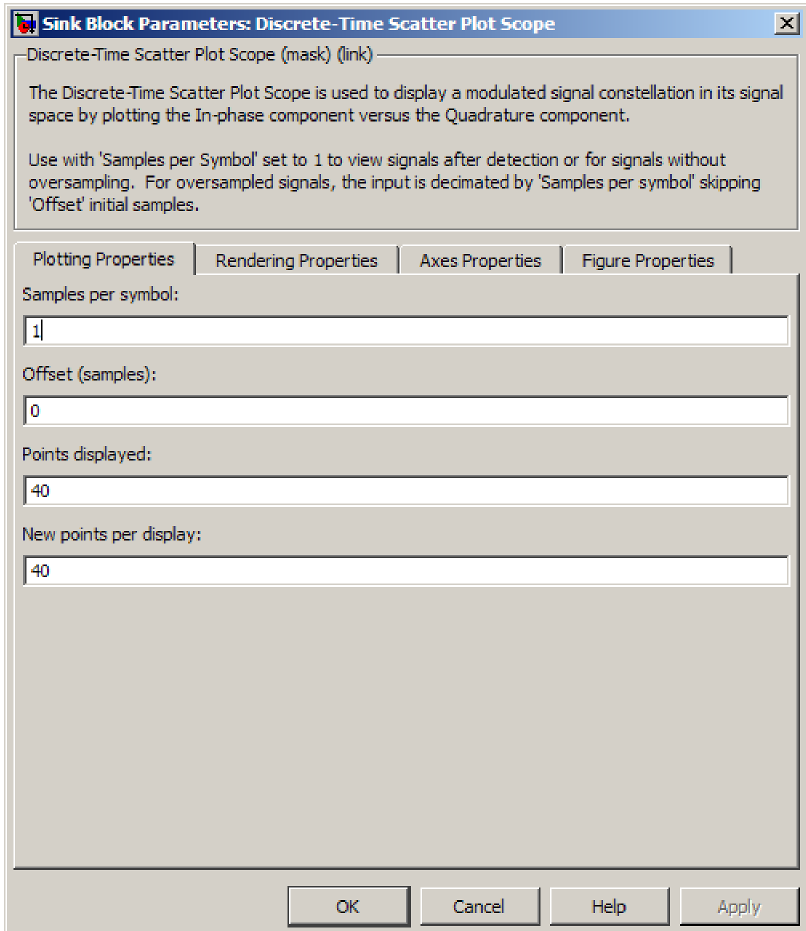
Note Before running a model that contains a Discrete-Time Scatter Plot Scope block in Accelerator, Rapid Accelerator, or External mode, you must select **Open scope at start of simulation**. If you do not select this check box before running your model for the first time, the scope will not display your simulation data

Behavior in Enabled Subsystems

You can use the Discrete-Time Scatter Plot Scope block inside an enabled subsystem. However, you cannot use the scope block inside an enabled subsystem when the model is in a multirate multitasking environment.

When you use the scope in a multirate singletasking environment, it may generate unexpected results inside enabled subsystems. To workaround this issue, configure the scope for single-rate mode. See “Recommended Settings” on page 2-208 for the parameter settings that enable single-rate mode.

Dialog Box



Samples per symbol

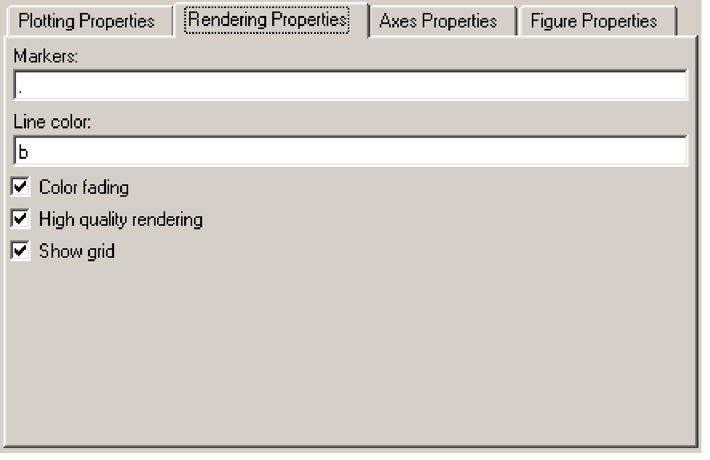
Number of samples per symbol.

Offset (samples)

Nonnegative integer less than the number of samples per symbol, specifying the number of samples to skip before plotting points.

Points displayed
Total number of points plotted.

New points per display
Number of new points that appear in each display.



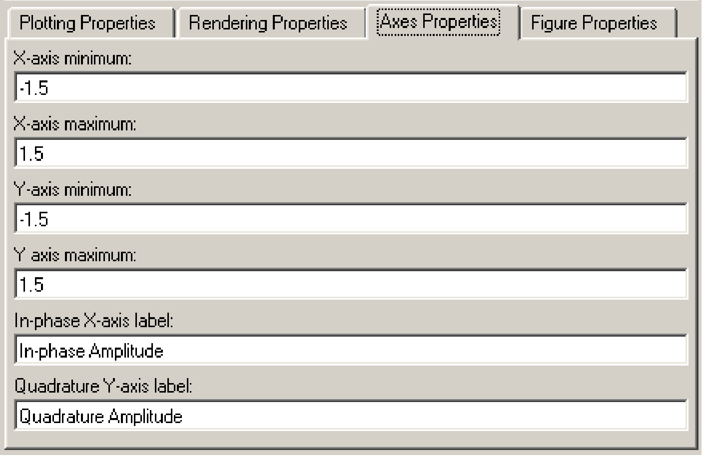
Markers
Line markers used in the scatter plot. Tunable.

Line color
The line color used in the scatter plot. Tunable.

Color fading
When selected, the points in the scatter plot fade as the interval of time after they are first plotted increases. Tunable.

High quality rendering
When selected, the block renders a slow, higher-quality picture with overwrite raster operations. When cleared, the block renders a fast, lower-quality picture with XOR raster operations. Tunable.

Show grid
Toggles the scope grid on and off. Tunable.



X-axis minimum
: Minimum value the scope displays on the *x*-axis. Tunable.

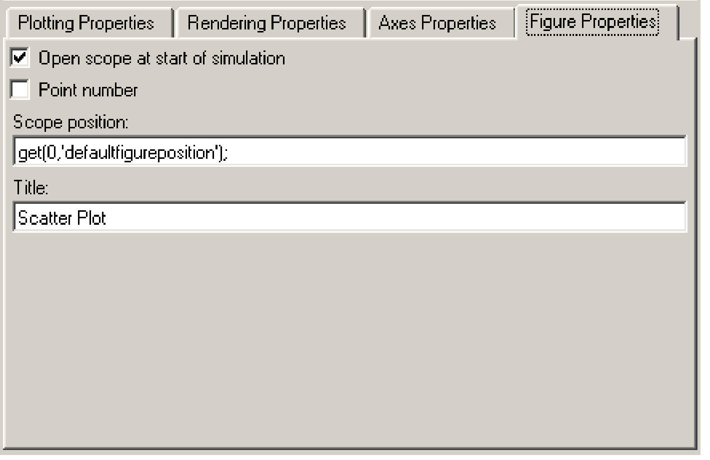
X-axis maximum
: Maximum value the scope displays on the *x*-axis. Tunable.

Y-axis minimum
: Minimum signal value the scope displays on the *y*-axis. Tunable.

Y-axis maximum
: Maximum signal value the scope displays on the *y*-axis. Tunable.

In-phase X-axis label
: Label for *x*-axis. Tunable.

Quadrature Y-axis label
: Label for *y*-axis. Tunable.



Open at start of simulation

When selected, the scope opens at the start of simulation. When cleared, you must double-click the block after the start of simulation to open the scope.

Note Before running a model that contains a Discrete-Time Scatter Plot Scope block in Accelerator, Rapid Accelerator, or External mode, you must select **Open scope at start of simulation**. If you do not select this check box before running your model for the first time, the scope will not display your simulation data

---

Point number

Displays the number of the current point in the input sequence. Tunable.

Scope position

A four-element vector of the form [left bottom width height] specifying the position of the scope window. (0,0) is the lower left corner of the display. Tunable.

Title

Title of scatter plot. Tunable.

Examples

For documentation examples that use this block, see "View a Sinusoid" and "View a Modulated Signal".

The following demos in Communications System Toolbox software illustrate how to use the Discrete-Time Scatter Plot Scope block:

- Digital Video Broadcasting Model — Terrestrial
- HiperLAN/2
- Phase Noise Effects in 256 QAM
- Multipath Rayleigh Fading Channel

See Also

Discrete-Time Eye Diagram Scope, Discrete-Time Signal Trajectory Scope, Real-Imag to Complex

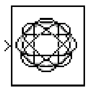
Purpose

Plot modulated signal's in-phase component versus its quadrature component

Library

Comm Sinks

Description



The Discrete-Time Signal Trajectory Scope displays the trajectory of a modulated signal in its signal space by plotting its in-phase component versus its quadrature component.

The Discrete-Time Signal Trajectory Scope block has one input port. This block accepts a complex scalar-valued or column vector input signal. The block accepts a signal with the following data types: `double`, `single`, base integer, and fixed-point for input, but will cast it as `double`.

Line Style and Color

The **Line style** and **Line color** parameters on the **Rendering Properties** panel control the appearance of the signal trajectory. The **Line style** parameter specifies the style for lines in the signal trajectory. For details on the options for these parameters, see the reference page for the Discrete-Time Eye Diagram Scope block.

Recommended Settings

The following table summarizes the recommended parameter settings for the Discrete-Time Signal Trajectory Scope.

Parameter	Recommended Setting
Samples per symbol	Same as the **Samples per symbol** setting in the modulator block, or the **Interpolation factor** used in the interpolation block
Symbols displayed	10 times the alphabet size of the modulator, M

Parameter	Recommended Setting
New symbols per display	Same as **Symbols displayed** for greater speed A small positive integer for best animation
Line style	Solid dash (-)
Line color	Blue (b)
Color fading	Check **Color fading** for animation that resembles an oscilloscope. Clear for greater speed and animation that resembles a plot.
High quality rendering	Check **High quality rendering** for higher quality rendering. Clear for greater speed.
Open at start of simulation	Check **Open at start of simulation** to view the signal at the start of simulation. Clear to view the signal after convergence to steady state and for greater initial speed.
Y-axis minimum	Approximately 10% less than the expected minimum value of the signal
Y-axis maximum	Approximately 10% greater than the expected maximum value of the signal

For Rapid Accelerator or External mode, set the scope up for single rate mode. To guarantee the satisfactory behavior of single rate mode, the subsystem below the block mask for this block must operate as a single-rate entity, which means the following conditions are true:

sps * nspd = Sf

where:

- sps = Samples per symbol
- nspd = New symbols per display
- Sf = Input frame size, in samples

This equation guarantees that the subsystem below the mask for this block operates as a single-rate entity.

Warning

If you want to use Rapid Accelerator or External mode, set this block up to run as a single rate entity because the block does not support multi-rate in these modes.

Note Before running a model that contains a Discrete-Time Signal Trajectory Scope block in Accelerator, Rapid Accelerator, or External mode, you must select **Open scope at start of simulation**. If you do not select this check box before running your model for the first time, the scope will not display your simulation data

---

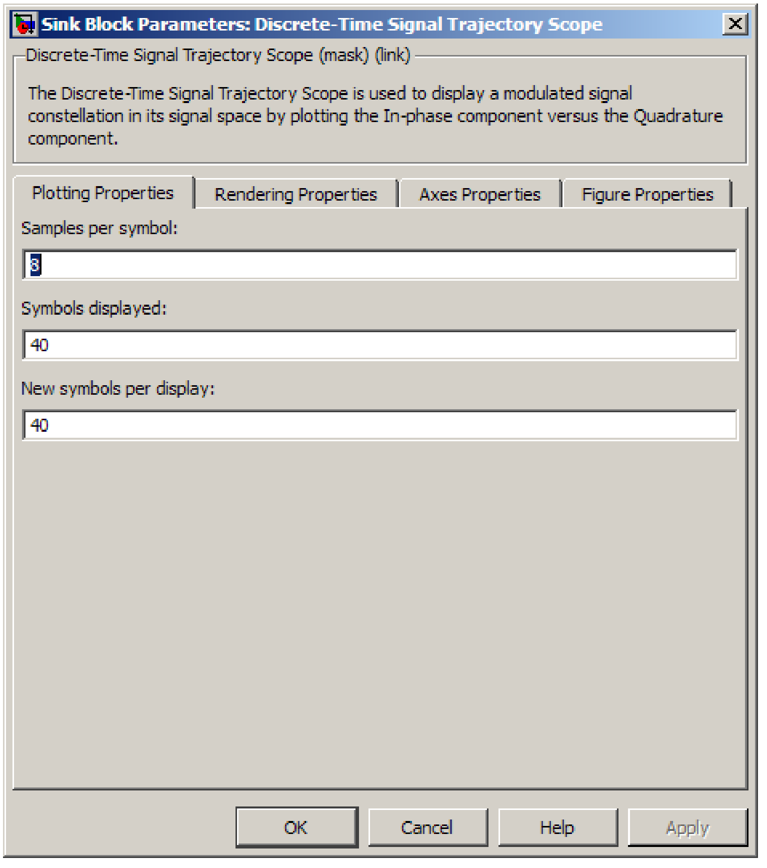
Behavior in Enabled Subsystems

You can use the Discrete-Time Signal Trajectory Scope block inside an enabled subsystem. However, you cannot use the scope block inside an enabled subsystem when the model is in a multirate multitasking environment.

When you use the scope in a multirate singletasking environment, it may generate unexpected results inside enabled subsystems. To workaround this issue, configure the scope for single-rate mode. See

"Recommended Settings" on page 2-217 for the parameter settings that enable single-rate mode.

Dialog Box

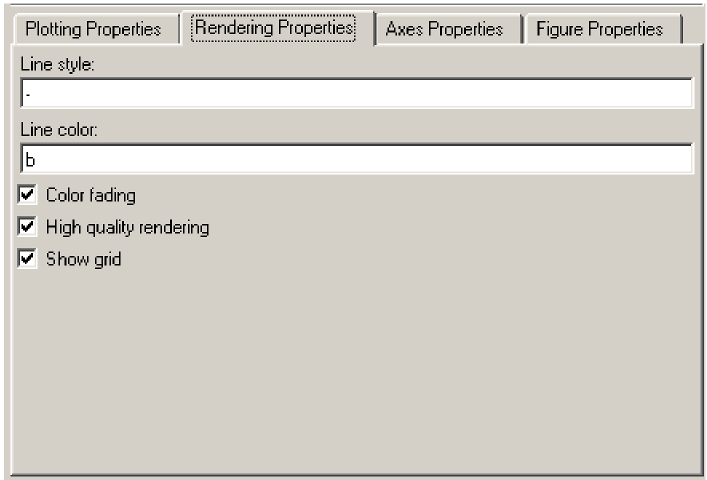


Samples per symbol
Number of samples per symbol.

Symbols displayed
Total number of symbols plotted.

New symbols per display

Number of new symbols that appear in each display.



Line markers

The line markers used in the signal trajectory. Tunable.

Line color

The line color used in the signal trajectory. Tunable.

Color fading

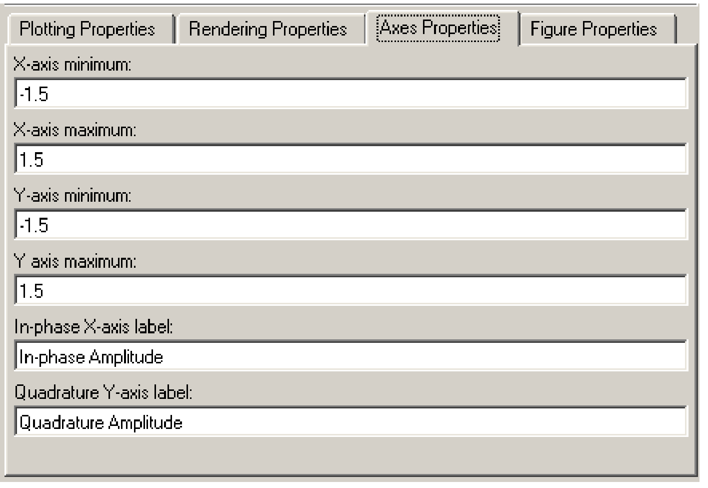
When selected, the points in the signal trajectory fade as the interval of time after they are first plotted increases. Tunable.

High quality rendering

When selected, the block renders a slow, higher-quality picture with overwrite raster operations. When cleared, the block renders a fast, lower-quality picture with XOR raster operations. Tunable.

Show grid

Toggles the scope grid on and off. Tunable.



X-axis minimum
Minimum value the scope displays on the x-axis. Tunable.

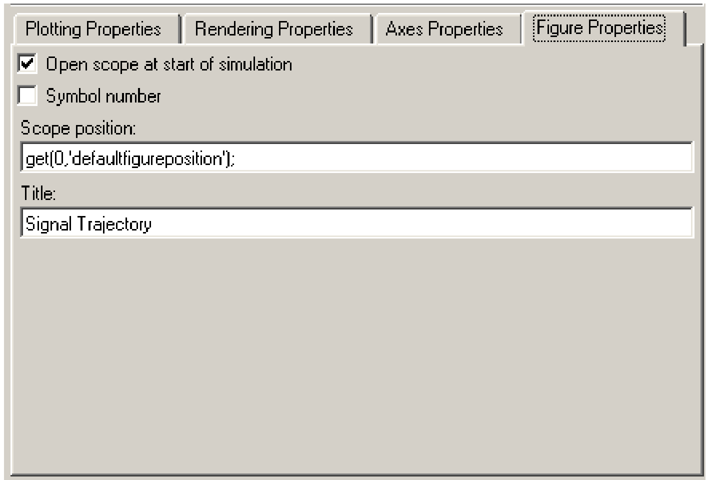
X-axis maximum
Maximum value the scope displays on the *x*-axis. Tunable.

Y-axis minimum
Minimum signal value the scope displays on the *y*-axis. Tunable.

Y-axis maximum
Maximum signal value the scope display on the *y*-axis. Tunable.

In-phase X-axis label
Label for *x*-axis. Tunable.

Quadrature Y-axis label
Label for *y*-axis. Tunable.



Open at start of simulation

When selected, the scope opens at the start of simulation. When cleared, you must double-click the block after the start of simulation to open the scope. Tunable

---

Note Before running a model that contains a Discrete-Time Signal Trajectory Scope block in Accelerator, Rapid Accelerator, or External mode, you must select **Open scope at start of simulation**. If you do not select this check box before running your model for the first time, the scope will not display your simulation data

---

Symbol number

Displays the number of the current symbol in the input sequence. Tunable.

Scope position

A four-element vector of the form [left bottom width height] specifying the position of the scope window. (0,0) is the lower left corner of the display. Tunable.

Title

Title of signal trajectory plot. Tunable.

Examples

For documentation examples that use this block, see "View a Sinusoid" and "View a Modulated Signal".

Also, the following demos in Communications System Toolbox software illustrate how to use the Discrete-Time Signal Trajectory Scope:

- Filtered Offset QPSK vs. Filtered QPSK
- GMSK vs. MSK

See Also

Discrete-Time Eye Diagram Scope, Discrete-Time Scatter Plot Scope

Discrete-Time VCO

Purpose

Implement voltage-controlled oscillator in discrete time

Library

Components sublibrary of Synchronization

Description

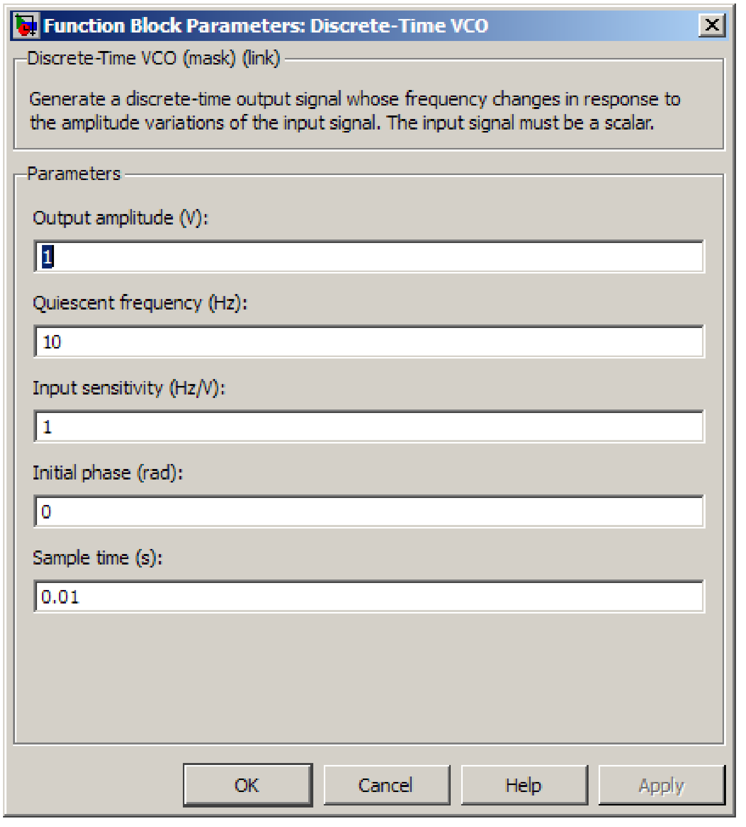
The Discrete-Time VCO (voltage-controlled oscillator) block generates a signal whose frequency shift from the **Quiescent frequency** parameter is proportional to the input signal. The input signal is interpreted as a voltage. If the input signal is $u(t)$, then the output signal is

$$y(t) = A_c \cos\left(2\pi f_c t + 2\pi k_c \int_0^t u(\tau)d\tau + \varphi\right)$$

where A_c is the **Output amplitude**, f_c is the **Quiescent frequency**, k_c is the **Input sensitivity**, and φ is the **Initial phase**

This block uses a discrete-time integrator to interpret the equation above.

This block accepts a scalar-valued input signal with a data type of `single` or `double`. The output signal inherits its data type from the input signal. The block supports double precision only for code generation.



Dialog Box

Output amplitude

The amplitude of the output.

Quiescent frequency (Hz)

The frequency of the oscillator output when the input signal is zero.

Input sensitivity

This value scales the input voltage and, consequently, the shift from the **Quiescent frequency** value. The units of **Input sensitivity** are Hertz per volt.

Initial phase (rad)
The initial phase of the oscillator in radians.

Sample time
The calculation sample time.

See Also

Continuous-Time VCO

DQPSK Demodulator Baseband

Purpose

Demodulate DQPSK-modulated data

Library

PM, in Digital Baseband sublibrary of Modulation

Description

The DQPSK Demodulator Baseband block demodulates a signal that was modulated using the differential quaternary phase shift keying method. The input is a baseband representation of the modulated signal.

The input must be a discrete-time complex signal. The output depends on the phase difference between the current symbol and the previous symbol. The first integer (or binary pair, if you set the **Output type** parameter to `Bit`) at the block output is the initial condition of zero because there is no previous symbol.

This block accepts either a scalar or column vector input signal. For information about the data types each block port supports, see "Supported Data Types" on page 2-230.

Outputs and Constellation Types

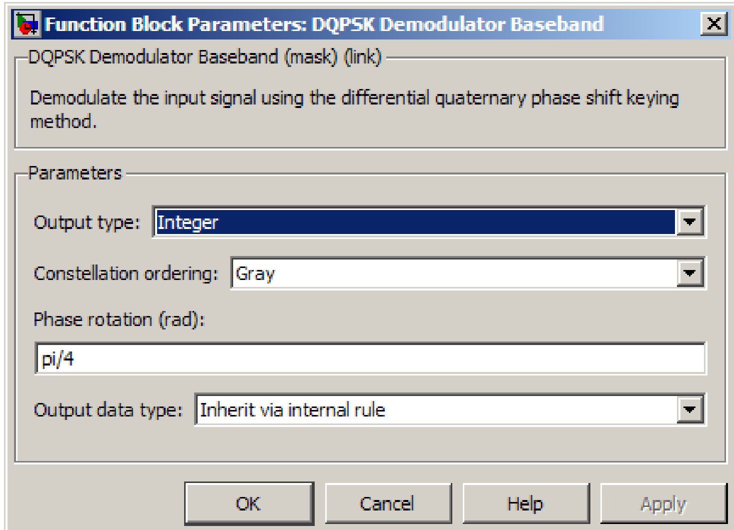
When you set **Output type** parameter to `Integer`, the block maps a phase difference of

$$\theta + \pi m/2$$

to m, where θ represents the **Phase rotation** parameter and m is 0, 1, 2, or 3.

When you set the **Output type** parameter to `Bit`, then the output contains pairs of binary values. The reference page for the DQPSK Modulator Baseband block shows which phase differences map to each binary pair, for the cases when the **Constellation ordering** parameter is either `Binary` or `Gray`.

Dialog Box



Output type

Determines whether the output consists of integers or pairs of bits.

Constellation ordering

Determines how the block maps each integer to a pair of output bits.

Phase rotation (rad)

This phase difference between the current and previous modulated symbols results in an output of zero.

Output data type

When the parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is of type `single` or `double`.

For integer outputs, this block can output the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`. For bit outputs, output can be `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, or `double`.

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 • Boolean when **Output type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

DQPSK Modulator Baseband

See Also

M-DPSK Demodulator Baseband, DBPSK Demodulator Baseband, QPSK Demodulator Baseband

Purpose

Modulate using differential quaternary phase shift keying method

Library

PM, in Digital Baseband sublibrary of Modulation

Description

The DQPSK Modulator Baseband block modulates using the differential quaternary phase shift keying method. The output is a baseband representation of the modulated signal.

The input must be a discrete-time signal. For information about the data types each block port supports, see "Supported Data Types" on page 2-235.

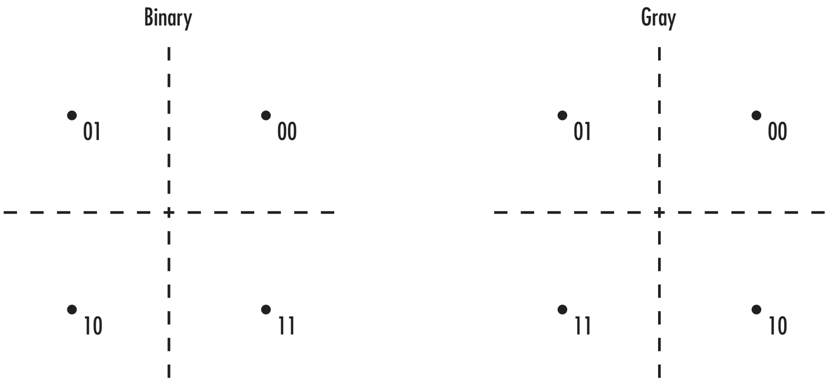
Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, the valid input values are 0, 1, 2, and 3. In this case, the block accepts a scalar or column vector input signal. If the first input is *m*, then the modulated symbol is

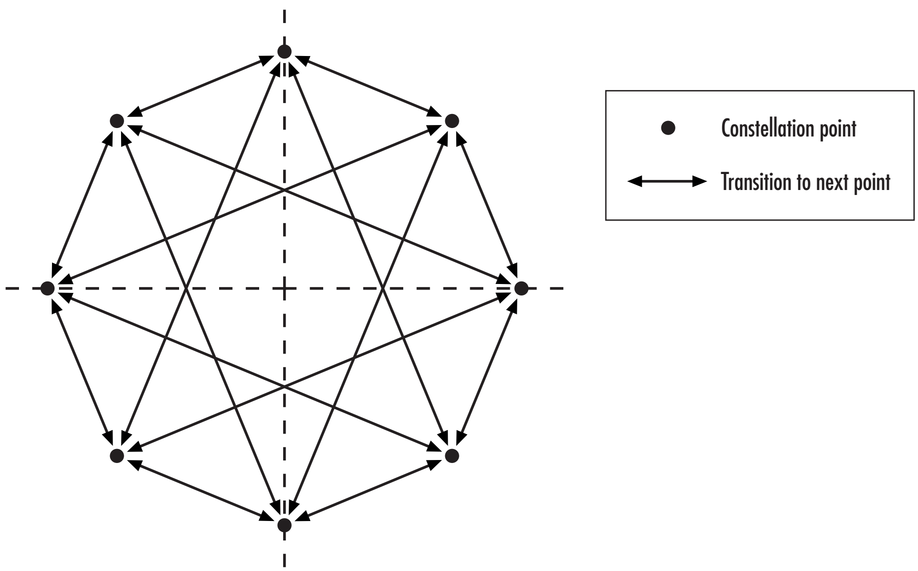
$$\exp(j\theta + j\pi m/2)$$

where θ represents the **Phase rotation** parameter. If a successive input is *m*, then the modulated symbol is the previous modulated symbol multiplied by $\exp(j\theta + j\pi m/2)$.

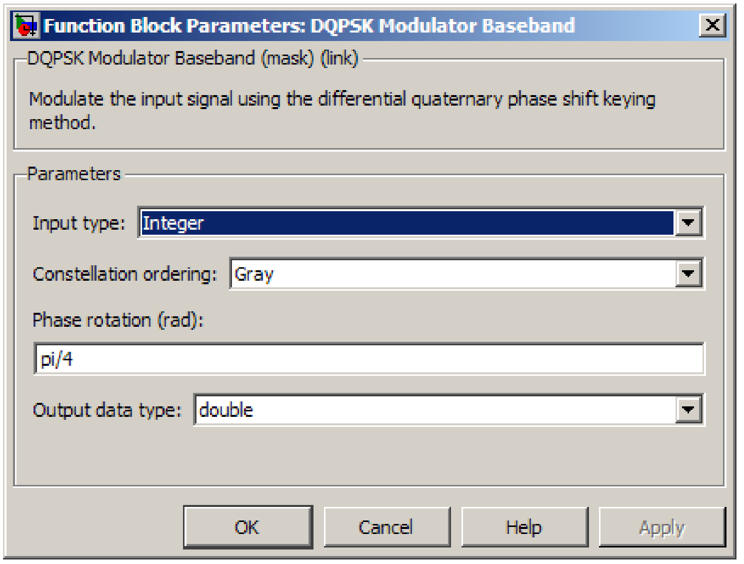
When you set the **Input type** parameter to `Bit`, the input contains pairs of binary values. In this case, the block accepts a column vector whose length is an even integer. The following figure shows the complex numbers by which the block multiples the previous symbol to compute the current symbol, depending on whether you set the **Constellation ordering** parameter to `Binary` or `Gray`. The following figure assumes that you set the **Phase rotation** parameter to $\frac{\Pi}{4}$; in other cases, the two schematics would be rotated accordingly.



The following figure shows the signal constellation for the DQPSK modulation method when you set the **Phase rotation** parameter to $\frac{\Pi}{4}$. The arrows indicate the four possible transitions from each symbol to the next symbol. The `Binary` and `Gray` options determine which transition is associated with each pair of input values.



More generally, if the **Phase rotation** parameter has the form $\frac{\Pi}{k}$ for some integer k, then the signal constellation has $2k$ points.



Dialog Box

Input type
: Indicates whether the input consists of integers or pairs of bits.

Constellation ordering
: Determines how the block maps each pair of input bits to a corresponding integer, using either a Binary or Gray mapping scheme.

Phase rotation (rad)
: The phase difference between the previous and current modulated symbols when the input is zero.

Output Data type
: The output data type can be either `single` or `double`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean when **Input type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point

Pair Block

DQPSK Demodulator Baseband

See Also

M-DPSK Modulator Baseband, DBPSK Modulator Baseband, QPSK Modulator Baseband

DSB AM Demodulator Passband

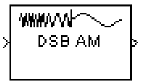
Purpose

Demodulate DSB-AM-modulated data

Library

Analog Passband Modulation, in Modulation

Description

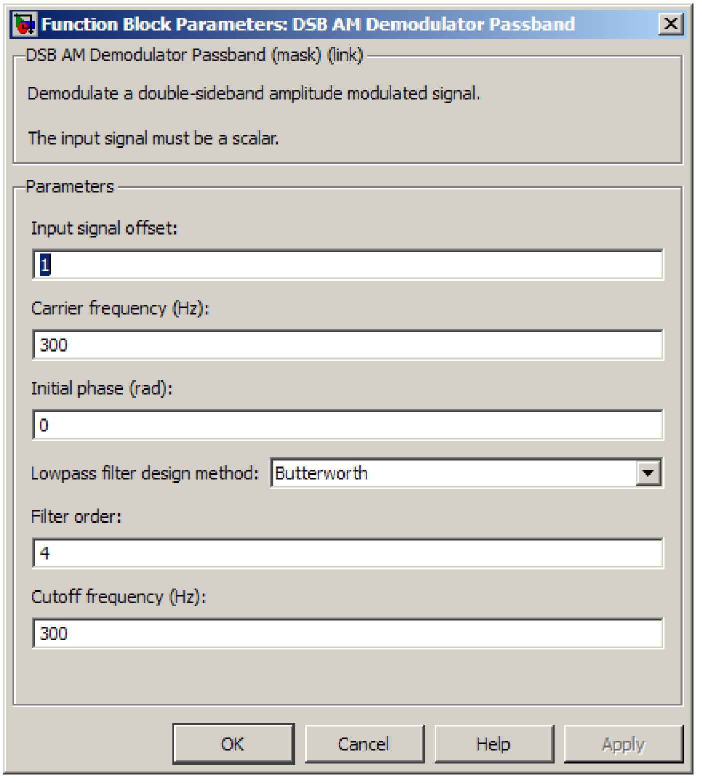


The DSB AM Demodulator Passband block demodulates a signal that was modulated using double-sideband amplitude modulation. The block uses the envelope detection method. The input is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

In the course of demodulating, this block uses a filter whose order, coefficients, passband ripple and stopband ripple are described by their respective lowpass filter parameters.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.



Dialog Box

Input signal offset

The same as the **Input signal offset** parameter in the corresponding DSB AM Modulator Passband block.

Carrier frequency (Hz)

The frequency of the carrier in the corresponding DSB AM Modulator Passband block.

Initial phase (rad)

The initial phase of the carrier in radians.

Lowpass filter design method
: The method used to generate the filter. Available methods are Butterworth, Chebyshev type I, Chebyshev type II, and Elliptic.

Filter order
: The order of the lowpass digital filter specified in the **Lowpass filter design method** field .

Cutoff frequency (Hz)
: The cutoff frequency of the lowpass digital filter specified in the **Lowpass filter design method** field in Hertz.

Passband ripple (dB)
: Applies to Chebyshev type I and Elliptic filters only. This is peak-to-peak ripple in the passband in dB.

Stopband ripple (dB)
: Applies to Chebyshev type II and Elliptic filters only. This is the peak-to-peak ripple in the stopband in dB.

Pair Block

DSB AM Modulator Passband

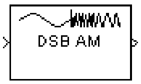
Purpose

Modulate using double-sideband amplitude modulation

Library

Analog Passband Modulation, in Modulation

Description



The DSB AM Modulator Passband block modulates using double-sideband amplitude modulation. The output is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

If the input is $u(t)$ as a function of time t, then the output is

$$(u(t) + k)\cos(2\pi f_c t + \theta)$$

where:

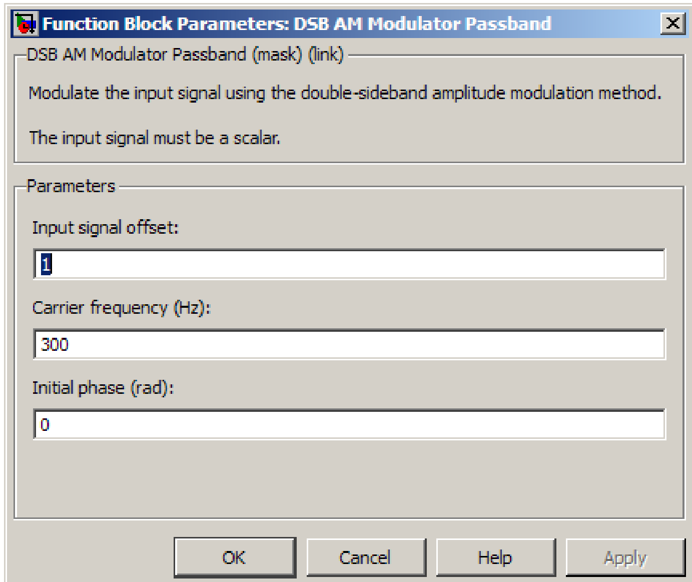
- k is the **Input signal offset** parameter.
- f_c is the **Carrier frequency** parameter.
- θ is the **Initial phase** parameter.

It is common to set the value of k to the maximum absolute value of the negative part of the input signal $u(t)$.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Dialog Box



Input signal offset

The offset factor k. This value should be greater than or equal to the absolute value of the minimum of the input signal.

Carrier frequency (Hz)

The frequency of the carrier.

Initial phase (rad)

The initial phase of the carrier.

Pair Block

DSB AM Demodulator Passband

See Also

DSBSC AM Modulator Passband, SSB AM Modulator Passband

Purpose

Demodulate DSBSC-AM-modulated data

Library

Analog Passband Modulation, in Modulation

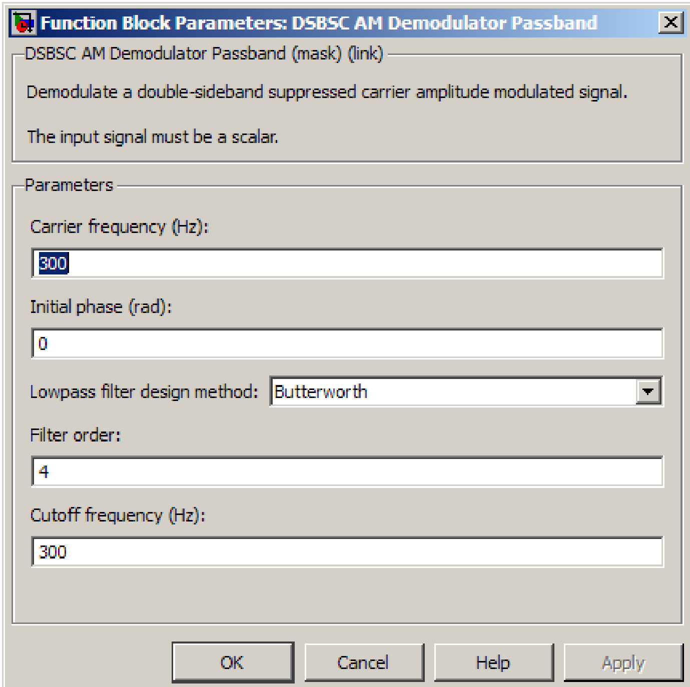
Description

The DSBSC AM Demodulator Passband block demodulates a signal that was modulated using double-sideband suppressed-carrier amplitude modulation. The input is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

In the course of demodulating, this block uses a filter whose order, coefficients, passband ripple and stopband ripple are described by the their respective lowpass filter parameters.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.



Dialog Box

Carrier frequency (Hz)
: The carrier frequency in the corresponding DSBSC AM Modulator Passband block.

Initial phase (rad)
: The initial phase of the carrier in radians.

Lowpass filter design method
: The method used to generate the filter. Available methods are Butterworth, Chebyshev type I, Chebyshev type II, and Elliptic.

Filter order
: The order of the lowpass digital filter specified in the **Lowpass filter design method** field .

Cutoff frequency (Hz)
The cutoff frequency of the lowpass digital filter specified in the Lowpass filter design method field in Hertz.

Passband Ripple (dB)
Applies to Chebyshev type I and Elliptic filters only. This is peak-to-peak ripple in the passband in dB.

Stopband Ripple (dB)
Applies to Chebyshev type II and Elliptic filters only. This is the peak-to-peak ripple in the stopband in dB.

Pair Block

DSBSC AM Modulator Passband

See Also

DSB AM Demodulator Passband, SSB AM Demodulator Passband

DSBSC AM Modulator Passband

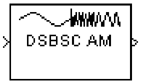
Purpose

Modulate using double-sideband suppressed-carrier amplitude modulation

Library

Analog Passband Modulation, in Modulation

Description



The DSBSC AM Modulator Passband block modulates using double-sideband suppressed-carrier amplitude modulation. The output is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

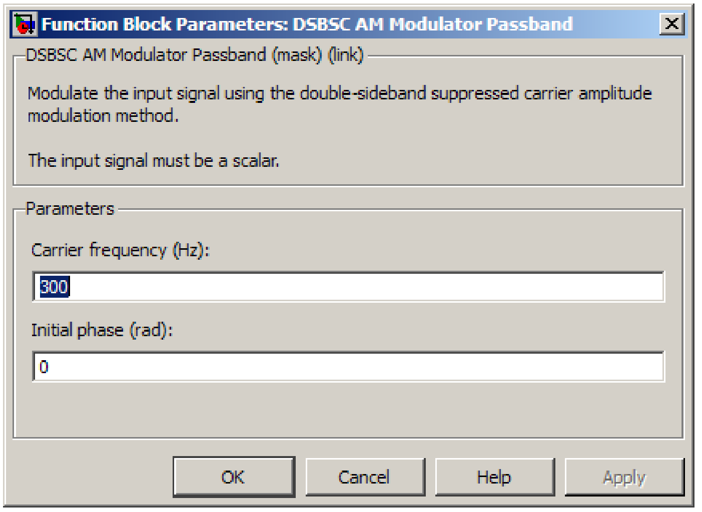
If the input is $u(t)$ as a function of time t, then the output is

$$u(t)\cos(2\pi f_c t + \theta)$$

where f_c is the **Carrier frequency** parameter and θ is the **Initial phase** parameter.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.



Dialog Box

Carrier frequency (Hz)
The frequency of the carrier.

Initial phase (rad)
The initial phase of the carrier in radians.

Pair Block

DSBSC AM Demodulator Passband

See Also

DSB AM Modulator Passband, SSB AM Modulator Passband

Early-Late Gate Timing Recovery

Purpose

Recover symbol timing phase using early-late gate method

Library

Timing Phase Recovery sublibrary of Synchronization

Description

The Early-Late Gate Timing Recovery block recovers the symbol timing phase of the input signal using the early-late gate method. This block implements a non-data-aided feedback method.

Inputs

By default, the block has one input port. Typically, the input signal is the output of a receive filter that is matched to the transmitting pulse shape.

This block accepts a scalar-valued or column vector input signal. The input uses N samples to represent each symbol, where $N > 1$ is the **Samples per symbol** parameter.

- For a column vector input signal, the block operates in single-rate processing mode. In this mode, the output signal inherits its sample rate from the input signal. The input length must be a multiple of N.
- For a scalar input signal, the block operates in multirate processing mode. In this mode, the input and output signals have different sample rates. The output sample rate equals N multiplied by the input sample rate.
- This block accepts input signals of type Double or Single

If you set the **Reset** parameter to `On nonzero input via port`, then the block has a second input port, labeled `Rst`. The `Rst` input determines when the timing estimation process restarts, and must be a scalar.

- If the input signal is a scalar value, the sample time of the `Rst` input equals the symbol period
- If the input signal is a column vector, the sample time of the `Rst` input equals the input port sample time
- This block accepts reset signals of type Double or Boolean

Outputs

The block has two output ports, labeled Sym and Ph:

- The Sym output is the result of applying the estimated phase correction to the input signal. This output is the signal value for each symbol, which can be used for decision purposes. The values in the Sym output occur at the symbol rate:
  - For a column vector input signal of length $N*R$, the Sym output is a column vector of length R having the same sample rate as the input signal.
  - For a scalar input signal, the sample rate of the Sym output equals N multiplied by the input sample rate.
- The Ph output gives the phase estimate for each symbol in the input.

  The Ph output contains nonnegative real numbers less than N. Noninteger values for the phase estimate correspond to interpolated values that lie between two values of the input signal. The sample time of the Ph output is the same as that of the Sym output.

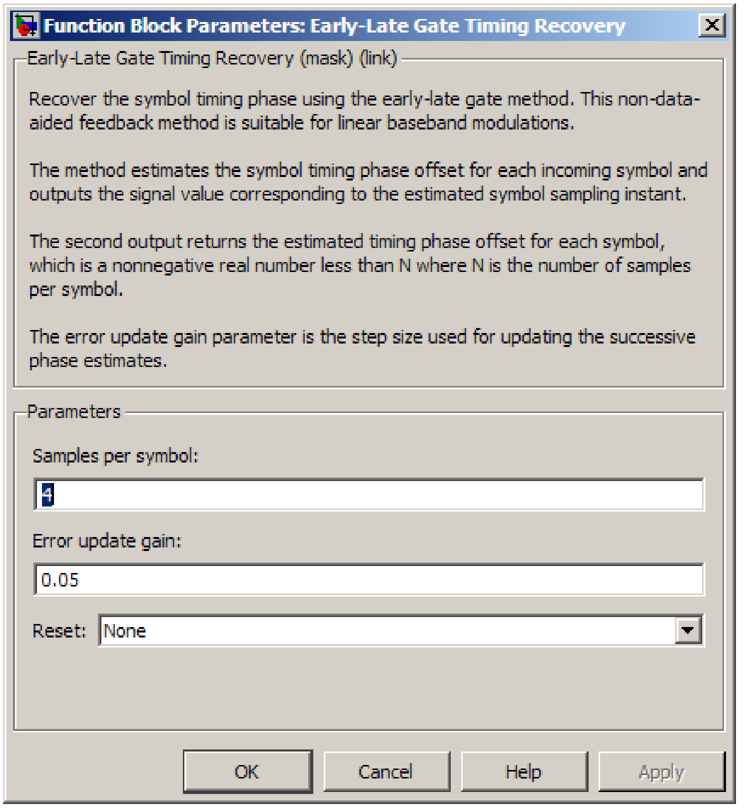
  Note If the Ph output is very close to either zero or **Samples per symbol**, or if the actual timing phase offset in your input signal is very close to zero, then the block's accuracy might be compromised by small amounts of noise or jitter. The block works well when the timing phase offset is significant rather than very close to zero.

- The output signal inherits its data type from the input signal.

Delays

When the input signal is a vector, this block incurs a delay of two symbols. When the input signal is a scalar, this block incurs a delay of three symbols.

Dialog Box



Samples per symbol

The number of samples, N, that represent each symbol in the input signal. This must be greater than 1.

Error update gain

A positive real number representing the step size that the block uses for updating successive phase estimates. Typically, this number is less than 1/N, which corresponds to a slowly varying phase.

This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode. If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. For more information, see Tunable Parameters in the Simulink *User's Guide*.

Reset

Determines whether and under what circumstances the block restarts the phase estimation process. Choices are `None`, `Every frame`, and `On nonzero input via port`. The last option causes the block to have a second input port, labeled Rst.

Algorithm

This block uses a timing error detector whose result for the kth symbol is e(k), given by

$$\begin{aligned} e(k) &= a_I(k) + a_Q(k) \\ a_I(k) &= y_I(kT + d_k)\left\{y_I(kT + T/2 + d_k) - y_I(kT - T/2 + d_{k-1})\right\} \\ a_Q(k) &= y_Q(kT + d_k)\left\{y_Q(kT + T/2 + d_k) - y_Q(kT - T/2 + d_{k-1})\right\} \end{aligned}$$

where

- y_I and y_Q are the in-phase and quadrature components, respectively, of the block's input signal
- T is the symbol period
- d_k is the phase estimate for the kth symbol

For more information about the role that e(k) plays in this block's algorithm, see "Feedback Methods for Timing Phase Recovery" in *Communications System Toolbox User's Guide*.

References

[1] Mengali, Umberto and Aldo N. D'Andrea, *Synchronization Techniques for Digital Receivers*, New York, Plenum Press, 1997.

[2] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, N.J., Prentice-Hall, 1988.

See Also

Gardner Timing Recovery, Squaring Timing Recovery, Mueller-Muller Timing Recovery

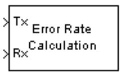
Purpose

Compute bit error rate or symbol error rate of input data

Library

Comm Sinks

Description



The Error Rate Calculation block compares input data from a transmitter with input data from a receiver. It calculates the error rate as a running statistic, by dividing the total number of unequal pairs of data elements by the total number of input data elements from one source.

Use this block to compute either symbol or bit error rate, because it does not consider the magnitude of the difference between input data elements. If the inputs are bits, then the block computes the bit error rate. If the inputs are symbols, then it computes the symbol error rate.

Note When you set the **Output data** parameter to `Workspace`, the block generates no code. If you need error rate information from generated code, set **Output data** to `Port`.

Input Data

This block has between two and four input ports, depending on how you set the dialog parameters. The inports marked `Tx` and `Rx` accept transmitted and received signals, respectively. The `Tx` and `Rx` signals must share the same sampling rate.

The `Tx` and `Rx` input ports accept scalar or column vector signals. For information about the data types each block port supports, see the "Supported Data Types" on page 2-260 table on this page. If `Tx` is a scalar and `Rx` is a vector, or vice-versa, then the block compares the scalar with each element of the vector. (Overall, the block behaves as if you had preprocessed the scalar signal with the Communications System Toolbox Repeat block with the **Rate options** parameter set to `Enforce single rate`.)

If you select **Reset port**, then an additional input port appears, labeled `Rst`. The `Rst` input accepts only a scalar signal (of type `double` or boolean) and must have the same port sample time as the Tx and Rx ports. When the Rst input is nonzero, the block clears and then recomputes the error statistics.

If you set the **Computation mode** parameter to Select samples from port, then an additional input port appears, labeled Sel. The Sel input indicates which elements of a frame are relevant for the computation. The Sel input can be a column vector of type double.

The guidelines below indicate how you should configure the inputs and the dialog parameters depending on how you want this block to interpret your Tx and Rx data.

- If both data signals are scalar, then this block compares the Tx scalar signal with the Rx scalar signal. For this configuration, use the **Computation mode** parameter default value, Entire frame.
- If both data signals are vectors, then this block compares some or all of the Tx and Rx data:
  - If you set the **Computation mode** parameter to Entire frame, then the block compares all of the Tx frame with all of the Rx frame.
  - If you set the **Computation mode** parameter to Select samples from mask, then the **Selected samples from frame** field appears in the dialog. This parameter field accepts a vector that lists the indices of those elements of the Rx frame that you want the block to consider. For example, to consider only the first and last elements of a length-six receiver frame, set the **Selected samples from frame** parameter to [1 6]. If the **Selected samples from frame** vector includes zeros, then the block ignores them.
  - If you set the **Computation mode** parameter to Select samples from port, then an additional input port, labeled Sel, appears on the block icon. The data at this input port must have the same format as that of the **Selected samples from frame** parameter described above.
- If one data signal is a scalar and the other is a vector, then this block compares the scalar with each entry of the vector. The three

subbullets above are still valid for this mode, except that if `Rx` is a scalar, then the phrase "`Rx` frame" above refers to the vector expansion of `Rx`.

Note This block does not support variable-size signals. If you choose the `Select samples from port` option and want the number of elements in the subframe to vary during the simulation, then you should pad the `Sel` signal with zeros. The Error Rate Calculation block ignores zeros in the `Sel` signal.

Output Data

This block produces a vector of length three, whose entries correspond to:

- The error rate
- The total number of errors, that is, the number of instances that an Rx element does not match the corresponding Tx element
- The total number of comparisons that the block made

The block sends this output data to the base MATLAB workspace or to an output port, depending on how you set the **Output data** parameter:

- If you set the **Output data** parameter to `Workspace` and fill in the **Variable name** parameter, then that variable in the base MATLAB workspace contains the current value when the simulation *ends*. Pausing the simulation does not cause the block to write interim data to the variable.

  If you plan to use this block along with the Simulink Coder software, then you should not use the `Workspace` option. Instead, use the `Port` option and connect the output port to a Simulink To Workspace block.
- If you set the **Output data** parameter to `Port`, then an output port appears. This output port contains the *running* error statistics.

Delays

The **Receive delay** and **Computation delay** parameters implement two different types of delays for this block. One delay is useful if you want this block to compensate for the delay in the received signal. The other is useful if you want to ignore the initial transient behavior of both input signals.

- The **Receive delay** parameter represents the number of samples by which the received data lags behind the transmitted data. The transmit signal is implicitly delayed by that same amount before the block compares it to the received data. This value is helpful when you delay the transmit signal so that it aligns with the received signal. The receive delay persists throughout the simulation.
- The **Computation delay** parameter represents the number of samples the block ignores at the beginning of the comparison.

If you do not know the receive delay in your model, you can use the Align Signals block, which automatically compensates for the delay. If you use the Align Signals block, set the **Receive delay** in the Error Rate Calculation block to `0` and the **Computation delay** to the value coming out of the `Delay` port of the Align Signals block.

Alternatively, you can use the Find Delay block to find the value of the delay, and then set the **Receive delay** parameter in the Error Rate Calculation block to the delay value.

If you use the `Select samples from mask` or `Select samples from port` option, then each delay parameter refers to the number of samples that the block receives, whether the block ultimately ignores some of them or not.

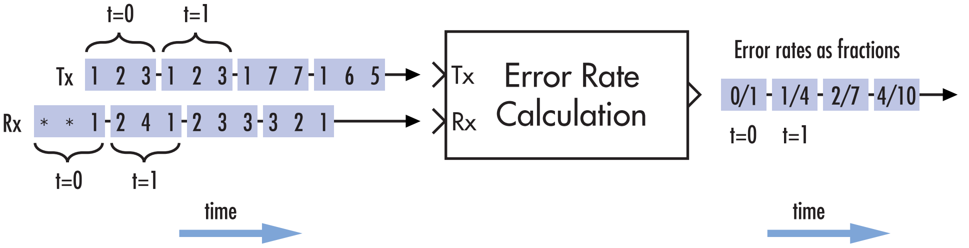
Stopping the Simulation Based on Error Statistics

You can configure this block so that its error statistics control the duration of simulation. This is useful for computing reliable steady-state error statistics without knowing in advance how long transient effects might last. To use this mode, check **Stop simulation**. The block attempts to run the simulation until it detects the number of errors the **Target number of errors** parameter specifies. However, the simulation stops before detecting enough errors if the time reaches the model's **Stop time** setting (in the **Configuration Parameters** dialog box), if the Error Rate Calculation block makes **Maximum number of symbols** comparisons, or if another block in the model directs the simulation to stop.

To ignore either of the two stopping criteria in this block, set the corresponding parameter (**Target number of errors** or **Maximum number of symbols**) to `Inf`. For example, to reach a target number of errors without stopping the simulation early, set **Maximum number of symbols** to `Inf` and set the model's **Stop time** to `Inf`.

Examples

The figure below shows how the block compares pairs of elements and counts the number of error events. The `Tx` and `Rx` inputs are column vectors.



This example assumes that the sample time of each input signal is 1 second and that the block's parameters are as follows:

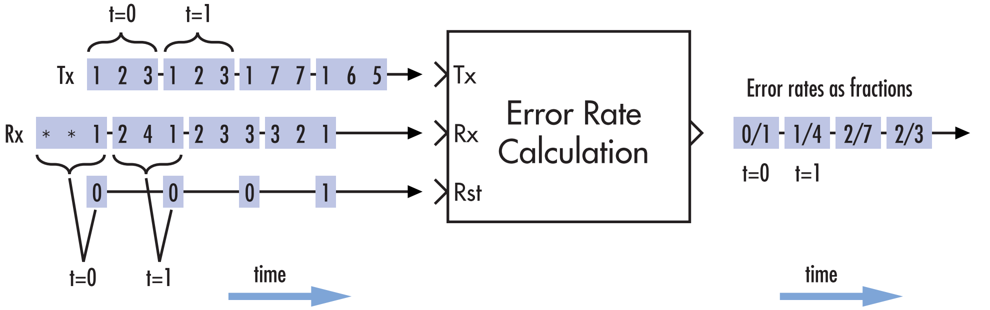
- **Receive delay** = `2`
- **Computation delay** = `0`
- **Computation mode** = `Entire frame`

Both input signals are column vectors of length three. However, the schematic arranges each column vector horizontally and aligns pairs of vectors so as to reflect a receive delay of two samples. At each time step, the block compares elements of the `Rx` signal with those of the `Tx` signal that appear directly above them in the schematic. For instance, at time 1, the block compares 2, 4, and 1 from the `Rx` signal with 2, 3, and 1 from the `Tx` signal.

The values of the first two elements of `Rx` appear as asterisks because they do not influence the output. Similarly, the 6 and 5 in the `Tx` signal do not influence the output up to time 3, though they *would* influence the output at time 4.

In the error rates on the right side of the figure, each numerator at time *t* reflects the number of errors when considering the elements of `Rx` up through time *t*.

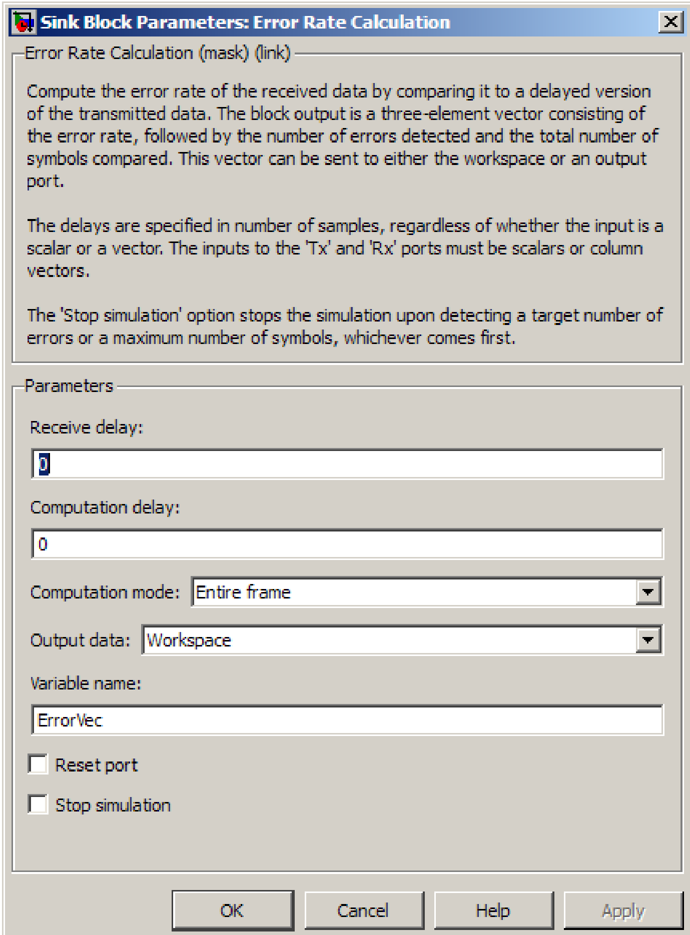
If the block's **Reset port** box had been checked and a reset had occurred at time = 3 seconds, then the last error rate would have been 2/3 instead of 4/10. This value 2/3 would reflect the comparison of 3, 2, and 1 from the `Rx` signal with 7, 7, and 1 from the `Tx` signal. The figure below illustrates this scenario. The `Tx` and `Rx` inputs are column vectors.



Tuning Parameters in an RSim Executable (Simulink Coder Software)

If you use the Simulink Coder rapid simulation (RSim) target to build an RSim executable, then you can tune the **Target number of errors** and **Maximum number of symbols** parameters without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

Dialog Box



Receive delay

Number of samples by which the received data lags behind the transmitted data. (If `Tx` or `Rx` is a vector, then each entry represents a sample.)

Computation delay
Number of samples that the block should ignore at the beginning of the comparison.

Computation mode
Either `Entire frame`, `Select samples from mask`, or `Select samples from port`, depending on whether the block should consider all or only part of the input frames.

Selected samples from frame
A vector that lists the indices of the elements of the `Rx` frame vector that the block should consider when making comparisons. This field appears only if **Computation mode** is set to `Select samples from mask`.

Output data
Either `Workspace` or `Port`, depending on where you want to send the output data.

Variable name
Name of variable for the output data vector in the base MATLAB workspace. This field appears only if **Output data** is set to `Workspace`.

Reset port
If you check this box, then an additional input port appears, labeled `Rst`.

Stop simulation
If you check this box, then the simulation runs only until this block detects a specified number of errors or performs a specified number of comparisons, whichever comes first.

Target number of errors
The simulation stops after detecting this number of errors. This field is active only if **Stop simulation** is checked.

Maximum number of symbols
The simulation stops after making this number of comparisons. This field is active only if **Stop simulation** is checked.

Supported Data Types

Port	Supported Data Types
Tx	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Rx	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Sel	• Double-precision floating point
Reset	• Double-precision floating point • Boolean

See Also

Align Signals, Find Delay

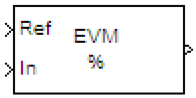
Purpose

Calculate vector magnitude difference between ideal reference signal and measured signal

Library

Utility Blocks

Description



EVM Measurement

Error Vector Magnitude (EVM) is a measurement of modulator or demodulator performance in an impaired signal.

While certain mask selections can change EVM block behavior, the block always has two input signals: a reference signal (at the reference port, `Ref`) and a corrupted signal (at the input port, `In`). You must select which normalization method the block uses when performing EVM calculations and which calculations you want as outputs.

The block either normalizes to the average reference signal power, average constellation power, or peak constellation power. For RMS EVM, Max EVM, and X-percentile EVM, the output computations reflect the normalization method.

The default EVM output is RMS EVM in percent, with an option of `Output maximum EVM` or `Output X-percentile EVM` values. The maximum EVM represents the worst-case EVM value per burst. For the X-percentile option, you can select to output the number of symbols processed in the percentile computations.

The following table shows the output type, the activation (what selects the output computation), computation units, and the corresponding computation duration.

Output	Activation	Units	Computation Duration
RMS EVM	Default	Percentage	Per burst
Max EVM	Parameter setting	Percentage	Per burst

Output	Activation	Units	Computation Duration
Percentile EVM	Parameter setting	Percentage	Continuous
Number of symbols	Parameter setting if you select **Output X-percentile EVM**	None	Continuous

The computation duration in per burst mode spans the symbols in the present burst. The computation duration in continuous mode spans all the symbols across multiple bursts.

Dimension

The block computes measurements for bursts of data. The data must be of length N, where N is the size of the burst. When computing RMS EVM or Max EVM, the block computes a unique output for each incoming burst; therefore, the computation duration is per burst.

The block computes the X-percentile for all incoming symbols across multiple bursts. This computation duration is the continuous mode (in contrast to the per burst duration for RMS EVM or Max EVM).

Input Signals

This block accepts scalar-valued or column vector input signals. The input and reference signals must have identical dimensions.

Output Signals

The output is always a scalar value.

Data Type

The block accepts double, single, and fixed-point data types. The output of the block is always double type.

Algorithms

The EVM block provides three different normalization methods. You can normalize measurements according to the average power of the reference signal, average constellation power, or peak constellation power. Different industry standards follow one of these normalization methods.

The following table lists how the block calculates the RMS EVM value for different normalization methods.

EVM Normalization Method	Algorithm
Reference Signal	$EVM_{RMS} = \sqrt{\dfrac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{\frac{1}{N}\sum_{k=1}^{N}(I_k^2 + Q_k^2)}} * 100$
Average Power	$EVM_{RMS} = \sqrt{\dfrac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{avg}}} * 100$
Peak Power	$EVM_{RMS} = \sqrt{\dfrac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\max}}} * 100$

where,

$e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$

I_k = In-phase measurement of the *kth* symbol in the burst

Q_k = Quadrature phase measurement of the *kth* symbol in the burst

N = Input vector length

P_{avg} = The value for **Average constellation power**

P_{max} = The value for **Peak constellation power**

I_k and Q_k represent ideal (reference) values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbols.

The max EVM is the maximum EVM value in a frame or

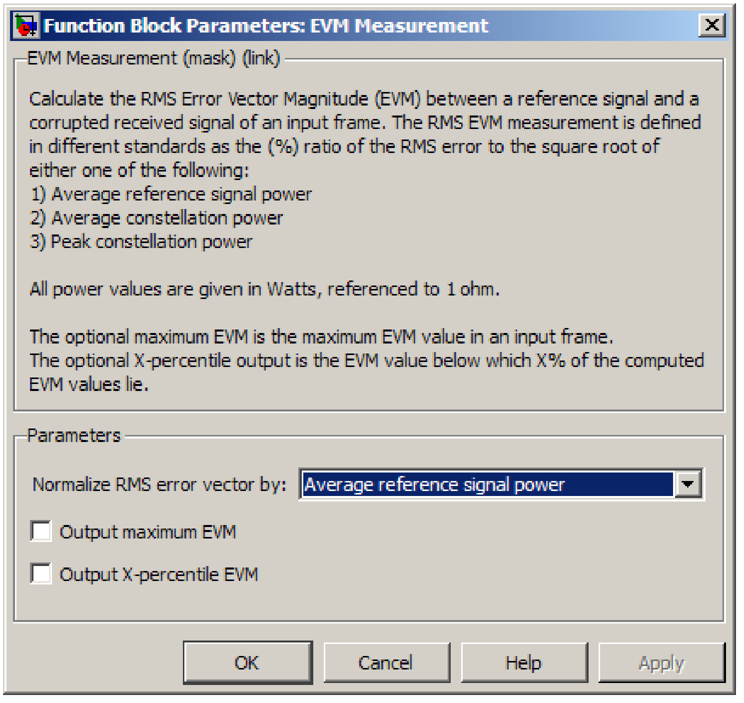
$$EVM_{\max} = \max_{k \in [1,\ldots,N]} \{EVM_k\}$$

where k is the kth symbol in a burst of length N.

The definition for $\mathrm{EVM_k}$ varies depending upon which normalization method you select for computing measurements. The block supports the algorithms in the following table.

EVM Normalization	Algorithm
Reference Signal	$$EVM_k = \sqrt{\frac{e_k}{\frac{1}{N}\sum_{k=1}^{N}(I_k^2 + Q_k^2)}} * 100$$
Average Power	$$EVM_k = \sqrt{\frac{e_k}{P_{avg}}} * 100$$
Peak Power	$$EVM_k = \sqrt{\frac{e_k}{P_{\max}}} * 100$$

The block computes X-percentile EVM by creating a histogram of all the incoming EVM_k values. The output provides the EVM value below which X% of the EVM values lay.



Dialog Box

Normalize RMS error vector by:

Selects the method by which the block normalizes measurements:

- `Average reference signal power`
- `Average constellation power`
- `Peak constellation power`

This parameter defaults to `Average reference signal power`.

Average constellation power:

Normalizes EVM measurement by the average constellation power. This parameter only appears if you set **Normalize RMS error vector** to `Average constellation power`.

Peak constellation power:
Normalizes EVM measurement by the peak constellation power. This parameter only appears if you set **Normalize RMS error vector** to `Peak constellation power`.

Output maximum EVM
Outputs the maximum EVM of an input vector or frame.

Output X-percentile EVM
Enables an output X-percentile EVM measurement. When you select this option, specify **X-percentile value (%)**.

X-percentile value (%)
This parameter only appears when you select **Output X-percentile EVM**. The Xth percentile is the EVM value below which X% of all the computed EVM values lie. The parameter defaults to the 95th percentile. Therefore, 95% of all EVM values are below this output.

Output the number of symbols processed
Outputs the number of symbols that the block uses to compute the **Output X-percentile EVM**. This parameter only appears when you select **Output X-percentile EVM**.

Examples

To see an example using the EVM block, refer to Measuring Modulator Accuracy in the Communications System Toolbox User's Guide.

References

References

[1] IEEE Standard 802.16-2004: “Part 16: Air interface for fixed broadband wireless access systems," October 2004. http://ieee802.org/16/published.html

[2] 3 GPP TS 45.005 V8.1.0 (2008-05): “Radio Access Network: Radio transmission and reception”

[3] IEEE Standard 802.11a-1999: “Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band,” 1999.

See Also

MER Measurement

Purpose

Find delay between two signals

Library

Utility Blocks

Description

The Find Delay block finds the delay between a signal and a delayed, and possibly distorted, version of itself. This is useful when you want to compare a transmitted and received signal to find the bit error rate, but do not know the delay in the received signal. This block accepts a column vector or matrix input signal. For a matrix input, the block outputs a row vector, and finds the delay in each channel of the matrix independently. See "Delays" for more information about signal delays.

The `sRef` input port receives the original signal, while the `sDel` input port receives the delayed version of the signal. The two input signals must have the same dimensions and sample times.

The output port labeled `delay` outputs the delay in units of samples. If you select `Include "change signal" output port`, then an output port labeled `chg` appears. The `chg` output port outputs 1 when there is a change from the delay computed at the previous sample, and 0 when there is no change. The `delay` output port outputs signals of type double, and the `chg` output port outputs signals of type boolean.

The block's **Correlation window length** parameter specifies how many samples of the signals the block uses to calculate the cross-correlation. The delay output is a nonnegative integer less than the **Correlation window length**.

As the **Correlation window length** is increased, the reliability of the computed delay also increases. However, the processing time to compute the delay increases as well.

You can make the Find Delay block stop updating the delay after it computes the same delay value for a specified number of samples. To do so, select **Disable recurring updates**, and enter a positive integer in the **Number of constant delay outputs to disable updates** field. For example, if you set **Number of constant delay outputs to disable updates** to `20`, the block will stop recalculating and updating the delay after it calculates the same value 20 times in succession.

Disabling recurring updates causes the simulation to run faster after the target number of constant delays occurs.

Tips for Using the Block Effectively

- Set **Correlation window length** sufficiently large so that the computed delay eventually stabilizes at a constant value. When this occurs, the signal from the optional `chg` output port stabilizes at the constant value of zero. If the computed delay is not constant, you should increase **Correlation window length**. If the increased value of **Correlation window length** exceeds the duration of the simulation, then you should also increase the duration of the simulation accordingly.
- If the cross-correlation between the two signals is broad, then the **Correlation window length** value should be much larger than the expected delay, or else the algorithm might stabilize at an incorrect value. For example, a CPM signal has a broad autocorrelation, so it has a broad cross-correlation with a delayed version of itself. In this case, the **Correlation window length** value should be much larger than the expected delay.
- If the block calculates a delay that is greater than 75 percent of the **Correlation window length**, the signal `sRef` is probably delayed relative to the signal `sDel`. In this case, you should switch the signal lines leading into the two input ports.

Examples

Finding the Delay Before Calculating an Error Rate

A typical use of this block is to determine the correct **Receive delay** parameter in the Error Rate Calculation block. This is illustrated in "Use the Find Delay and Align Signals Blocks". In that example, the modulation/demodulation operation introduces a computational delay into the received signal and the Find Delay block determines that the delay is 6 samples. This value of 6 becomes a parameter in the Error Rate Calculation block, which computes the bit error rate of the system.

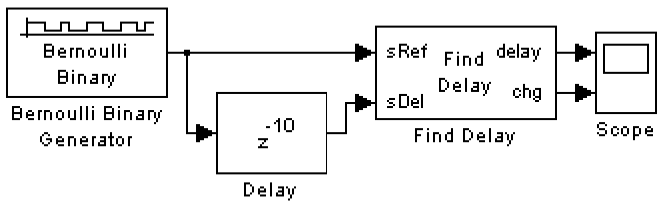
Another example of this usage is in "Delays".

Finding the Delay to Help Align Words

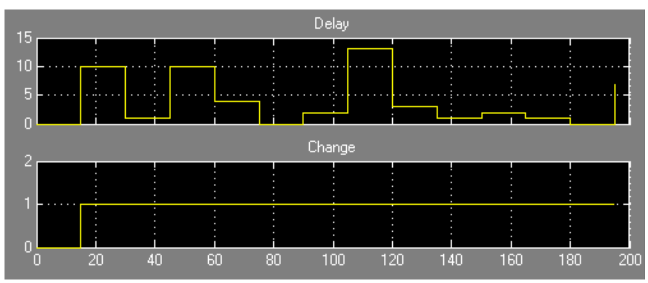
Another typical use of this block is to determine how to align the boundaries of frames with the boundaries of codewords or other types of data blocks. “Delays” describes when such alignment is necessary and also illustrates, in the “Aligning Words of a Block Code” discussion, how to use the Find Delay block to solve the problem.

Setting the Correlation Window Length

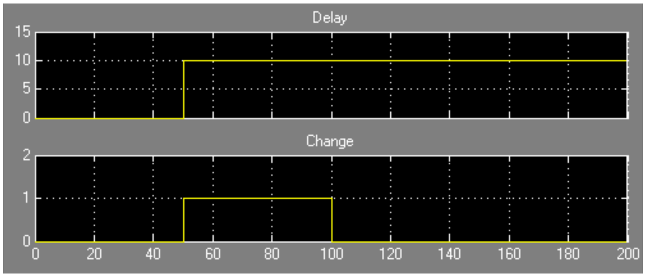
The next example illustrates how to tell when the **Correlation window length** is not sufficiently large.



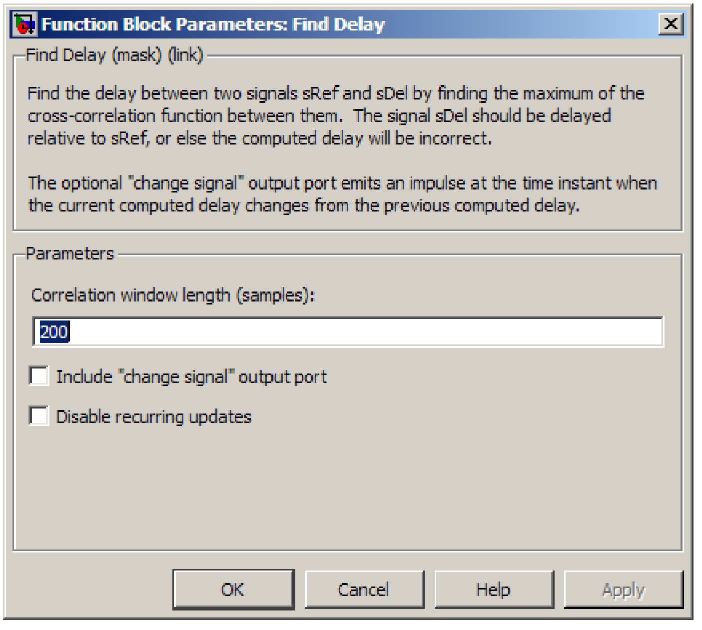
The model uses a Delay block to delay a signal by 10 samples. The Find Delay block compares the original signal with the delayed version. In this model, the **Input processing** parameter of the Delay block is set to `Columns as channels`. The model then displays the output of the Find Delay block in a scope. If the **Correlation window length** is `15`, the scope shows that the calculated delay is not constant over time, as you can see in the following image.



This result tells you to increase the **Correlation window length**. If you increase it to 50, the calculated delay stabilizes at 10, as shown below.



Dialog Box



Correlation window length
: The number of samples the block uses to calculate the cross-correlations of the two signals.

Include "change signal" output port
If you select this option, then the block has an extra output port that emits an impulse when the current computed delay differs from the previous computed delay.

Disable recurring updates
Selecting this option causes the block to stop computing the delay after it computes the same delay value for a specified number of samples.

Number of constant delay outputs to disable updates
A positive integer specifying how many times the block must compute the same delay before ceasing to update. This field appears only if **Disable recurring updates** is selected.

Algorithm

The Find Delay block finds the delay by calculating the cross-correlations of the first signal with time-shifted versions of the second signal, and then finding the index at which the cross-correlation is maximized.

See Also

Align Signals, Error Rate Calculation

FM Demodulator Passband

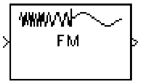
Purpose

Demodulate FM-modulated data

Library

Analog Passband Modulation, in Modulation

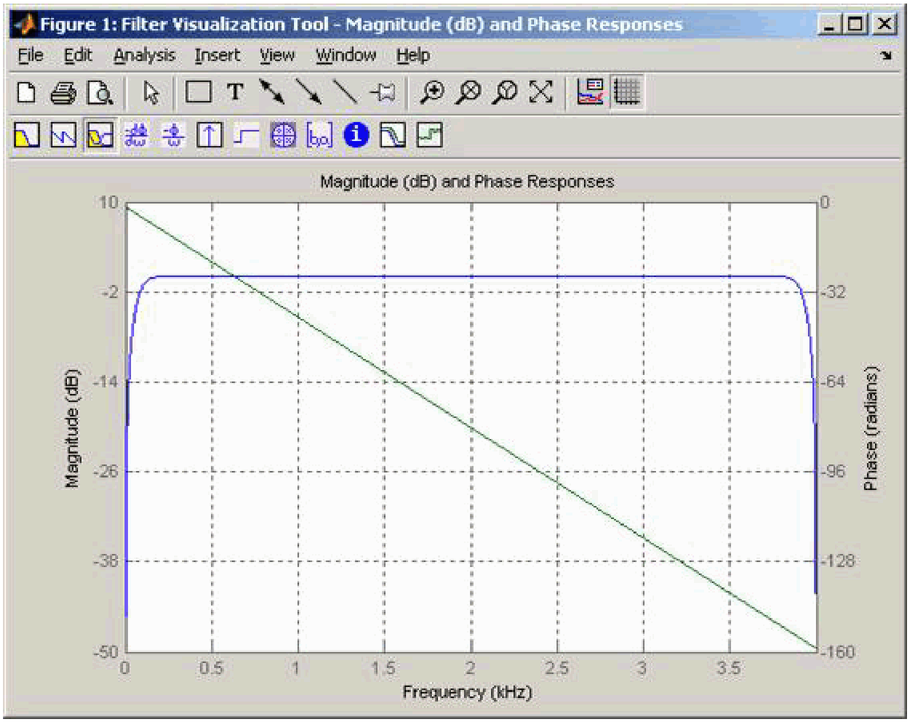
Description



The FM Demodulator Passband block demodulates a signal that was modulated using frequency modulation. The input is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

For best results, use a carrier frequency which is estimated to be larger than 10% of the reciprocal of your input signal's sample rate. This is due to the implementation of the Hilbert transform by means of a filter.

In the following example, we sample a 10Hz input signal at 8000 samples per second. We then designate a Hilbert Transform filter of order 100. Below is the response of the Hilbert Transform filter as returned by `fvtool`.

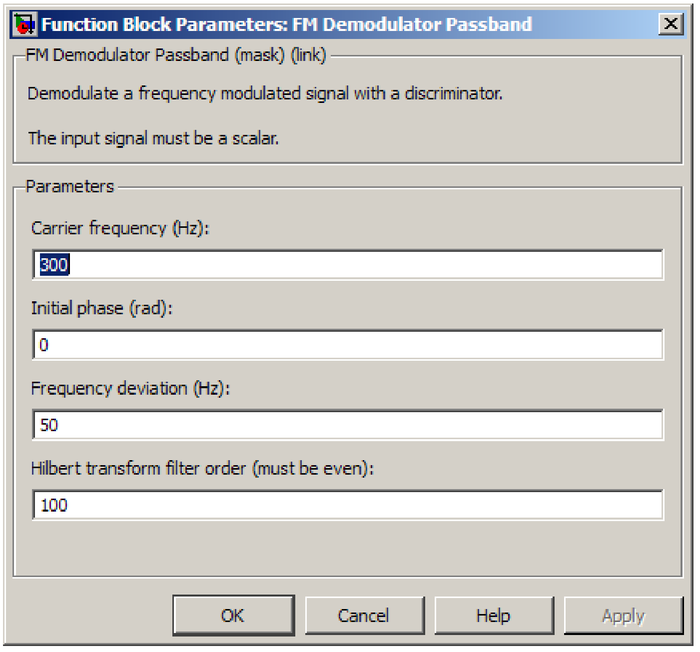


Note the bandwidth of the filter's magnitude response. By choosing a carrier frequency larger than 10% (but less than 90%) of the reciprocal of your input signal's sample time (8000 samples per second, in this example) or equivalently, a carrier frequency larger than 400Hz, we ensure that the Hilbert Transform Filter will be operating in the flat section of the filter's magnitude response (shown in blue), and that our modulated signal will have the desired magnitude and form.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Dialog Box



Carrier frequency (Hz)
: The frequency of the carrier.

Initial phase (rad)
: The initial phase of the carrier in radians.

Frequency deviation (Hz)
: The frequency deviation of the carrier frequency in Hertz. Sometimes it is referred to as the "variation" in the frequency.

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

Pair Block

FM Modulator Passband

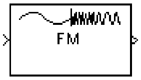
Purpose

Modulate using frequency modulation

Library

Analog Passband Modulation, in Modulation

Description



The FM Modulator Passband block modulates using frequency modulation. The output is a passband representation of the modulated signal. The output signal's frequency varies with the input signal's amplitude. Both the input and output signals are real scalar signals.

If the input is $u(t)$ as a function of time t, then the output is

$$\cos\left(2\pi f_c t + 2\pi K_c \int_0^t u(\tau) d\tau + \theta\right)$$

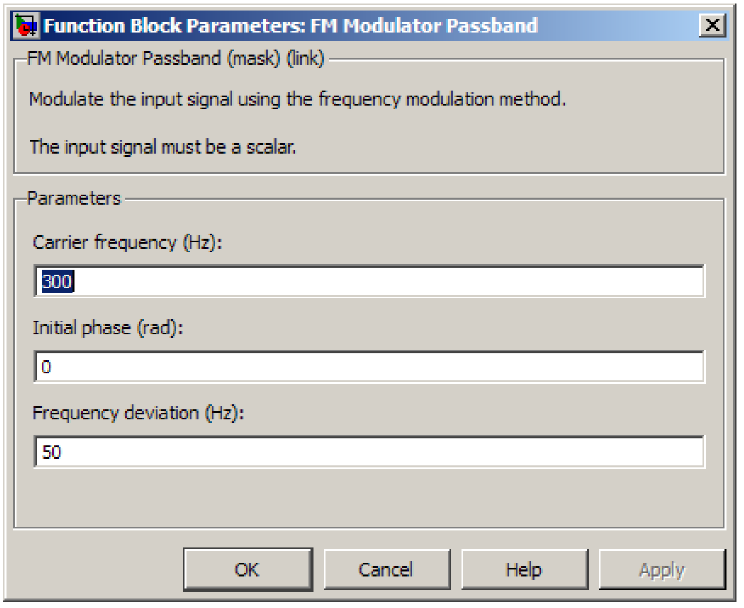
where:

- f_c represents the **Carrier frequency** parameter.
- θ represents the **Initial phase** parameter.
- K_c represents the **Frequency deviation** parameter.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal.

By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.



Dialog Box

Carrier frequency (Hz)
: The frequency of the carrier.

Initial phase (rad)
: The initial phase of the carrier in radians.

Frequency deviation (Hz)
: The frequency deviation of the carrier frequency in Hertz. Sometimes it is referred to as the "variation" in the frequency.

Pair Block

FM Demodulator Passband

Purpose

Reduce amplitude of input signal by amount specified

Library

RF Impairments

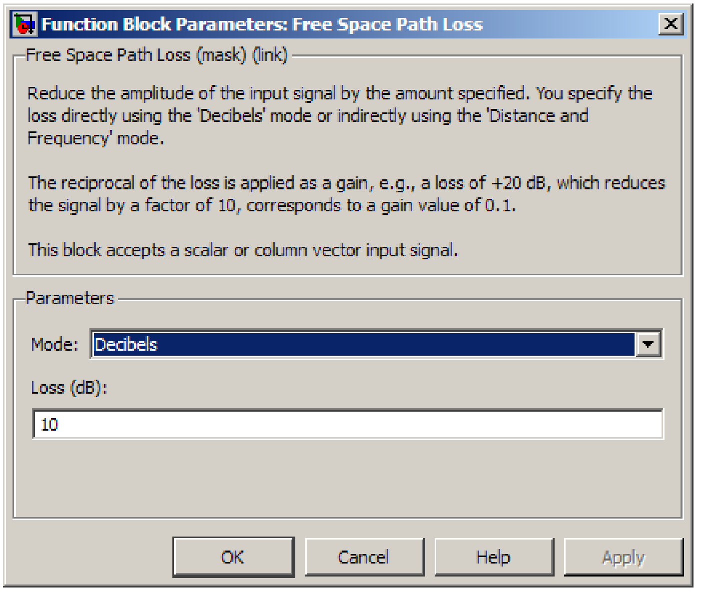
Description

The Free Space Path Loss block simulates the loss of signal power due to the distance between transmitter and receiver. The block reduces the amplitude of the input signal by an amount that is determined in either of two ways:

- By the **Distance (km)** and **Carrier frequency (MHz)** parameters, if you specify `Distance and Frequency` in the **Mode** field
- By the **Loss (dB)** parameter, if you specify `Decibels` in the **Mode** field

This block accepts a column vector input signal. The input signal to this block must be a complex signal.

Dialog Box



Mode
: Method of specifying the amount by which the signal power is reduced. The choices are `Decibels` and `Distance and Frequency`.

Loss
: The signal loss in decibels. This parameter appears when you set **Mode** to `Decibels`. The decibel amount shown on the mask is rounded for display purposes only.

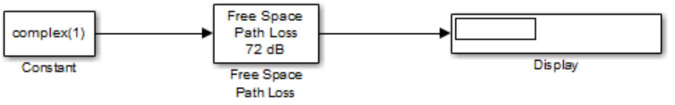
Distance
: Distance between transmitter and receiver in kilometers. This parameter appears when you set **Mode** to `Distance and Frequency`.

Carrier frequency (MHz)
: The carrier frequency in megahertz. This parameter appears when you set **Mode** to `Distance and Frequency`.

Examples

The model below illustrates the effect of the Free Space Path Loss block with the following parameter settings:

- **Mode** is set to `Distance and Frequency`.
- **Distance (km)** is set to `0.5`
- **Carrier frequency (MHz)** is set to `180`



See Also

Memoryless Nonlinearity

Gardner Timing Recovery

Purpose

Recover symbol timing phase using Gardner's method

Library

Timing Phase Recovery sublibrary of Synchronization

Description

The Gardner Timing Recovery block recovers the symbol timing phase of the input signal using Gardner's method. This block implements a non-data-aided feedback method that is independent of carrier phase recovery. The timing error detector that forms part of this block's algorithm requires at least two samples per symbol, one of which is the point at which the decision can be made.

The recovery method estimates the symbol timing phase offset for each incoming symbol and outputs the signal value corresponding to the estimated symbol sampling instant.

The second output returns the estimated timing phase recovery offset for each symbol, which is a nonnegative real number less than N, where N is the number of samples per symbol.

The error update gain parameter is the step size used for updating the successive phase estimates.

Inputs

By default, this block has one input port. Typically, the input signal is the output of a receive filter that is matched to the transmitting pulse shape. For best results, the input signal power should be less than 1.

This block accepts a scalar-valued or column vector input signal. The input uses N samples to represent each symbol, where $N > 1$ is the **Samples per symbol** parameter.

- For a column vector input signal, the block operates in single-rate processing mode. In this mode, the output signal inherits its sample rate from the input signal. The input length must be a multiple of N.
- For a scalar input signal, the block operates in multirate processing mode. In this mode, the input and output signals have different sample rates. The output sample rate equals N multiplied by the input sample rate.

- This block accepts input signals of type Double or Single

If you set the **Reset** parameter to `On nonzero input via port`, then the block has a second input port, labeled `Rst`. The `Rst` input determines when the timing estimation process restarts, and must be a scalar.

- If the input signal is a scalar value, the sample time of the `Rst` input equals the symbol period
- If the input signal is a column vector, the sample time of the `Rst` input equals the input port sample time
- This block accepts reset signals of type Double or Boolean

Outputs

The block has two output ports, labeled `Sym` and `Ph`:

- The `Sym` output is the result of applying the estimated phase correction to the input signal. This output is the signal value for each symbol, which can be used for decision purposes. The values in the `Sym` output occur at the symbol rate:
  - For a column vector input signal of length $N*R$, the `Sym` output is a column vector of length R having the same sample rate as the input signal.
  - For a scalar input signal, the sample rate of the `Sym` output equals N multiplied by the input sample rate.
- The `Ph` output gives the phase estimate for each symbol in the input.

  The `Ph` output contains nonnegative real numbers less than N. Noninteger values for the phase estimate correspond to interpolated values that lie between two values of the input signal. The sample time of the `Ph` output is the same as that of the `Sym` output.

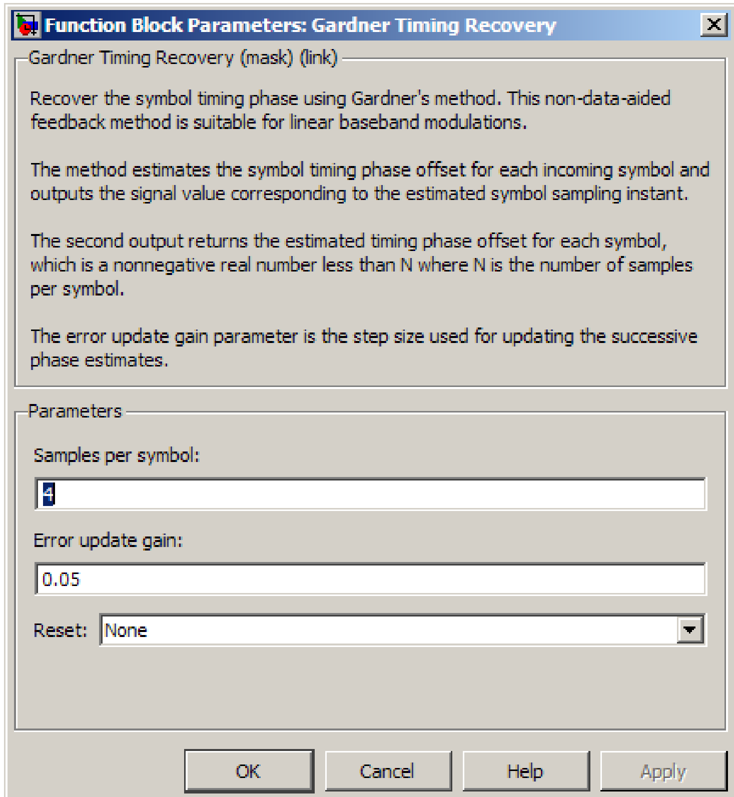
Note If the `Ph` output is very close to either zero or **Samples per symbol**, or if the actual timing phase offset in your input signal is very close to zero, then the block's accuracy might be compromised by small amounts of noise or jitter. The block works well when the timing phase offset is significant rather than very close to zero.

- The output signal inherits its data type from the input signal.

Delays

When the input signal is a vector, this block incurs a delay of two symbols. When the input signal is a scalar, this block incurs a delay of three symbols.

Dialog Box



Samples per symbol

The number of samples, N, that represent each symbol in the input signal. This must be greater than 1.

Error update gain

A positive real number representing the step size that the block uses for updating successive phase estimates. Typically, this number is less than 1/N, which corresponds to a slowly varying phase.

This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode. If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. For more information, see Tunable Parameters in the Simulink *User's Guide*.

Reset

Determines whether and under what circumstances the block restarts the phase estimation process. Choices are `None`, `Every frame`, and `On nonzero input via port`. The last option causes the block to have a second input port, labeled Rst.

Algorithm

This block uses a timing error detector whose result for the kth symbol is e(k), given by

$$e(k) = a_I(k) + a_Q(k)$$
$$a_I(k) = \left\{ y_I((k-1)T + d_{k-1}) - y_I(kT + d_k) \right\} y_I(kT - T/2 + d_{k-1})$$
$$a_Q(k) = \left\{ y_Q((k-1)T + d_{k-1}) - y_Q(kT + d_k) \right\} y_Q(kT - T/2 + d_{k-1})$$

where

- y_I and y_Q are the in-phase and quadrature components, respectively, of the block's input signal
- T is the symbol period
- d_k is the phase estimate for the kth symbol

Notice from the expressions in curly braces above that the timing error detector approximates the derivative of y using finite differences.

For more information about the role that e(k) plays in this block's algorithm, see "Feedback Methods for Timing Phase Recovery" in *Communications System Toolbox User's Guide*.

Supported Data Types

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

Examples

The `commgardnerphrecov` demonstration model uses this block.

References

[1] Gardner, F. M., "A BPSK/QPSK Timing-Error Detector for Sampled Receivers", *IEEE Transactions on Communications*, Vol. COM-34, No. 5, May 1986, pp. 423-429.

[2] Mengali, Umberto and Aldo N. D'Andrea, *Synchronization Techniques for Digital Receivers*, New York, Plenum Press, 1997.

[3] Meyr, Heinrich, Marc Moeneclaey, and Stefan A. Fechtel, *Digital Communication Receivers*, Vol 2, New York, Wiley, 1998.

[4] Oerder, M., "Derivation of Gardner's Timing-Error Detector from the ML principle", *IEEE Transactions on Communications*, Vol. COM-35, No. 6, June 1987, pp. 684-685.

See Also

Early-Late Gate Timing Recovery, Squaring Timing Recovery, Mueller-Muller Timing Recovery

Gaussian Filter

Purpose Filter input signal, possibly downsampling, using Gaussian FIR filter

Library Comm Filters

Description

The Gaussian Filter block filters the input signal using a Gaussian FIR filter. The block expects the input signal to be upsampled as its input, so that the **Input samples per symbol** parameter, N, is at least 2. The block's icon shows the filter's impulse response."

Characteristics of the Filter

The impulse response of the Gaussian filter is

$$h(t) = \frac{\exp\left(\dfrac{-t^2}{2\delta^2}\right)}{\sqrt{2\pi}\cdot\delta}$$

where

$$\delta = \frac{\sqrt{\ln(2)}}{2\pi BT}$$

and B is the filter's 3-dB bandwidth. The **BT product** parameter is B times the input signal's symbol period. For a given BT product, the Signal Processing Toolbox `gaussfir` function generates a filter that is half the bandwidth of the filter generated by the Communications System Toolbox Gaussian Filter block.

The **Group delay** parameter is the number of symbol periods between the start of the filter's response and the peak of the filter's response. The group delay and N determine the length of the filter's impulse response, which is 2 * N * **Group delay** + 1.

The **Filter coefficient normalization** parameter indicates how the block scales the set of filter coefficients:

- `Sum of coefficients` means that the sum of the coefficients equals 1.

- `Filter energy` means that the sum of the squares of the coefficients equals 1.
- `Peak amplitude` means that the maximum coefficient equals 1.

After the block normalizes the set of filter coefficients as above, it multiplies all coefficients by the **Linear amplitude filter gain** parameter. As a result, the output is scaled by $\sqrt{N}$. If the output of this block feeds the input to the AWGN Channel block, specify the AWGN signal power parameter to be 1/N.

Input and Output Signals

This block accepts scalar, column vector, and *M*-by-*N* matrix input signals. The block filters an *M*-by-*N* input matrix as follows:

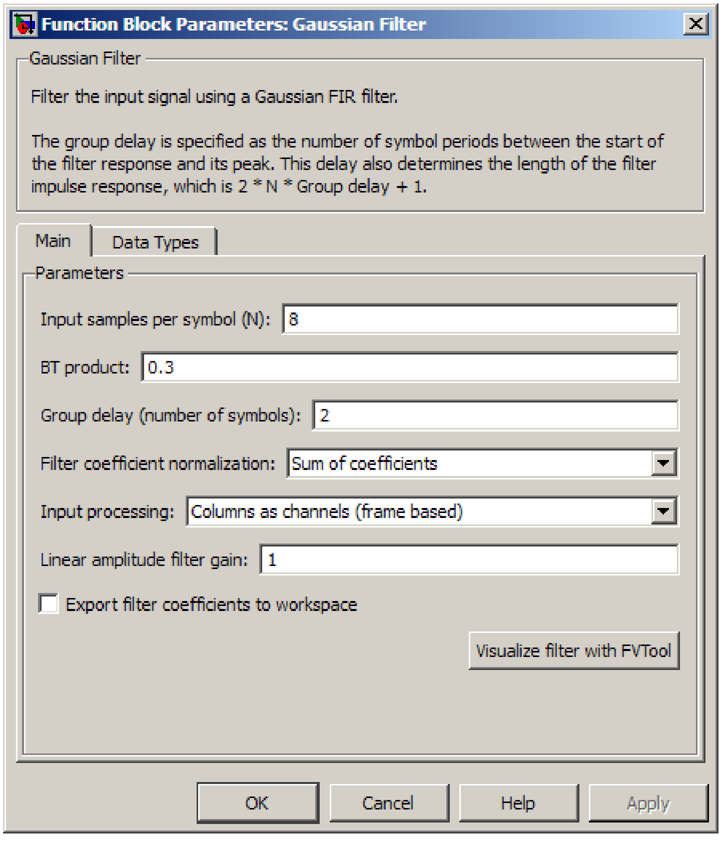
- When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each column as a separate channel. In this mode, the block creates *N* instances of the same filter, each with its own independent state buffer. Each of the *N* filters process *M* input samples 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. For information about the data types each block port supports, see the table on this page.

Exporting Filter Coefficients to the MATLAB Workspace

To examine or manipulate the coefficients of the filter that this block designs, select **Export filter coefficients to workspace**. Then set the **Coefficient variable name** parameter to the name of a variable that you want the block to create in the MATLAB workspace. Running the simulation causes the block to create the variable, overwriting any previous contents in case the variable already exists.

Dialog Box



Input samples per symbol

A positive integer representing the number of samples per symbol in the input signal.

BT product

The product of the filter's 3-dB bandwidth and the input signal's symbol period

Group delay

A positive integer that represents the number of symbol periods between the start of the filter response and its peak.

Filter coefficient normalization

The block scales the set of filter coefficients so that this quantity equals 1. Choices are `Sum of coefficients`, `Filter energy`, and `Peak amplitude`.

Input processing

Specify how the block processes the input signal. 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 *Communications System Toolbox Release Notes* for more information.

This parameter is available only when you set the **Rate options** parameter to `Allow multirate processing`.

Linear amplitude filter gain

A positive scalar used to scale the filter coefficients after the block uses the normalization specified in the **Filter coefficient normalization** parameter.

Export filter coefficients to workspace

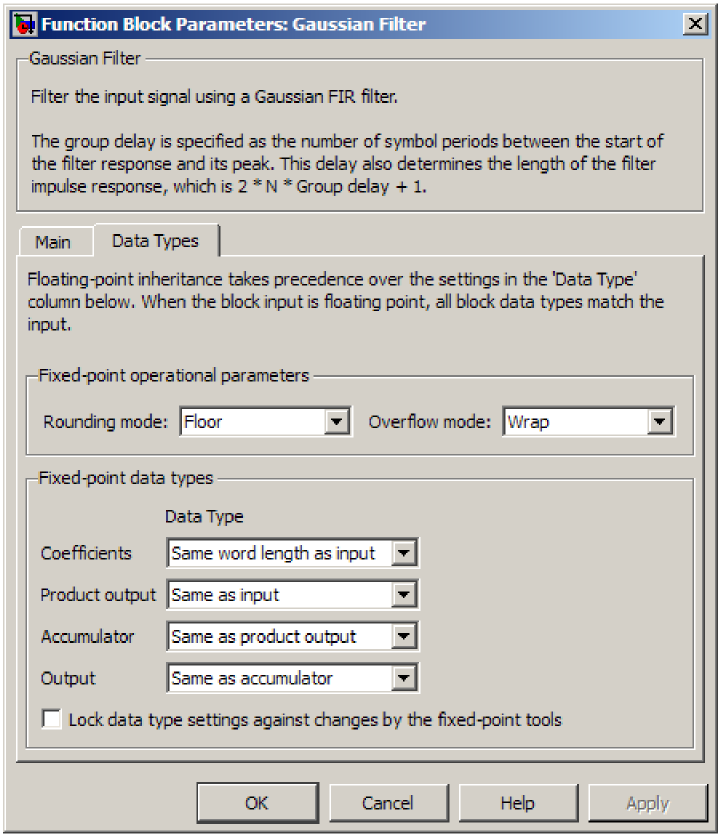
If you check this box, then the block creates a variable in the MATLAB workspace that contains the filter coefficients.

Coefficient variable name

The name of the variable to create in the MATLAB workspace. This field appears only if **Export filter coefficients to workspace** is selected.

Visualize filter with FVTool

If you click this button, then MATLAB launches the Filter Visualization Tool, `fvtool`, to analyze the Gaussian filter whenever you apply any changes to the block's parameters. If you launch `fvtool` for the filter, and subsequently change parameters in the mask, `fvtool` will not update. You will need to launch a new `fvtool` in order to see the new filter characteristics. Also note that if you have launched `fvtool`, then it will remain open even after the model is closed.



Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`. The block uses the **Rounding** selection if a number cannot be represented exactly by the specified data type and scaling, it is rounded to a representable number. For more information, see Rounding Modes in the DSP System Toolbox documentation or “Rounding Mode: Simplest” in the Fixed-Point Designer documentation.

Overflow mode

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

Coefficients

The block implementation uses a Direct-Form FIR filter. The **Coefficients** parameter controls which data type represents the coefficients when the input data is a fixed-point signal.

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). See "Filter Structure Diagrams" in the *DSP System Toolbox Reference Guide* 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 are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits. If applicable, you are able to enter separate fraction lengths for the numerator and denominator coefficients.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. If applicable, you are able to enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.

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

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See "Multiplication Data Types" and "Multiplication Data Types" in the *DSP System Toolbox Reference Guide* 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 are able to enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See "Filter Structure Diagrams" 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 are able to enter the word length and the fraction length of the accumulator, in bits.

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

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

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling tool in the Fixed-Point Tool.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Signed fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed fixed-point

See Also

Raised Cosine Receive Filter, `gaussfir`

References

[1] 3GPP TS 05.04 V8.4.0 — 3rd Generation Partnership Project; Technical Specification Group GSM/EDGE Radio Access Network; Digital cellular telecommunications system (Phase 2+); Modulation (Release 1999)

Gaussian Noise Generator

Purpose

Generate Gaussian distributed noise with given mean and variance values

Library

Noise Generators sublibrary of Comm Sources

Description

The Gaussian Noise Generator block generates discrete-time white Gaussian noise. You must specify the **Initial seed** vector in the simulation.

The **Mean Value** and the **Variance** can be either scalars or vectors. If either of these is a scalar, then the block applies the same value to each element of a sample-based output or each column of a frame-based output. Individual elements or columns, respectively, are uncorrelated with each other.

When the **Variance** is a vector, its length must be the same as that of the **Initial seed** vector. In this case, the covariance matrix is a diagonal matrix whose diagonal elements come from the **Variance** vector. Since the off-diagonal elements are zero, the output Gaussian random variables are uncorrelated.

When the **Variance** is a square matrix, it represents the covariance matrix. Its off-diagonal elements are the correlations between pairs of output Gaussian random variables. In this case, the **Variance** matrix must be positive definite, and it must be `N`-by-`N`, where `N` is the length of the **Initial seed**.

The probability density function of n-dimensional Gaussian noise is

$$f(x) = \left((2\pi)^n \det K\right)^{-1/2} \exp\left(-(x-\mu)^T K^{-1}(x-\mu)/2\right)$$

where x is a length-n vector, K is the n-by-n covariance matrix, μ is the mean value vector, and the superscript T indicates matrix transpose.

Initial Seed

The **Initial seed** parameter initializes the random number generator that the Gaussian Noise Generator block uses to add noise to the input signal. For best results, the **Initial seed** should be a prime number greater than 30. Also, if there are other blocks in a model that have an **Initial seed** parameter, you should choose different initial seeds for all such blocks.

You can choose seeds for the Gaussian Noise Generator block using the Communications System Toolbox randseed function. At the MATLAB prompt, enter

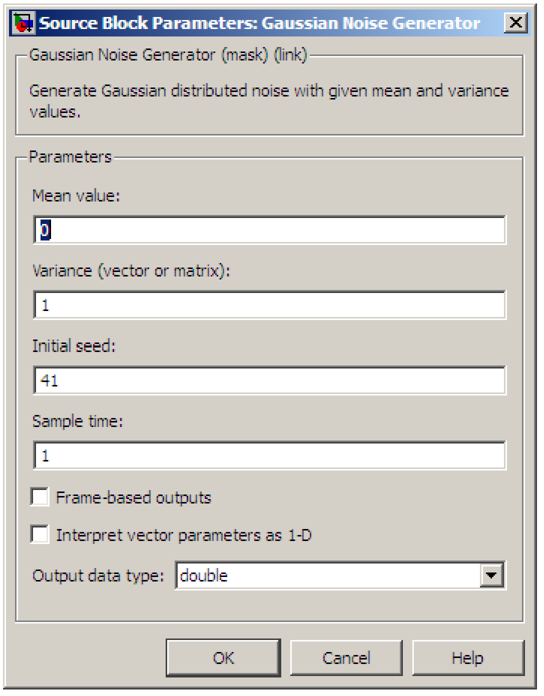
```
randseed
```

This returns a random prime number greater than 30. Entering `randseed` again produces a different prime number. If you supply an integer argument, `randseed` always returns the same prime for that integer. For example, `randseed(5)` always returns the same answer.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples per frame**, and **Interpret vector parameters as 1-D** parameters. See "Sources and Sinks" in the *Communications System Toolbox User's Guide* for more details.

If the **Initial seed** parameter is a vector, then its length becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. In this case, the shape (row or column) of the **Initial seed** parameter becomes the shape of a sample-based two-dimensional output signal. If the **Initial seed** parameter is a scalar but either the **Mean value** or **Variance** parameter is a vector, then the vector length determines the output attributes mentioned above.



Dialog Box

Mean value
The mean value of the random variable output.

Variance
The covariance among the output random variables.

Initial seed
The initial seed value for the random number generator.

Sample time
The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs
: Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame
: The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D
: If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type
: The output can be set to `double` or `single` data types.

See Also

Random Source (DSP System Toolbox documentation), AWGN Channel, `rand` (built-in MATLAB function), randseed

General Block Deinterleaver

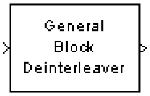
Purpose

Restore ordering of symbols in input vector

Library

Block sublibrary of Interleaving

Description



The General Block Deinterleaver block rearranges the elements of its input vector without repeating or omitting any elements. If the input contains N elements, then the **Elements** parameter is a column vector of length N. The column vector indicates the indices, in order, of the output elements that came from the input vector. That is, for each integer k between 1 and N,

$$\mathrm{Output}(\mathbf{Elements}(k)) = \mathrm{Input}(k)$$

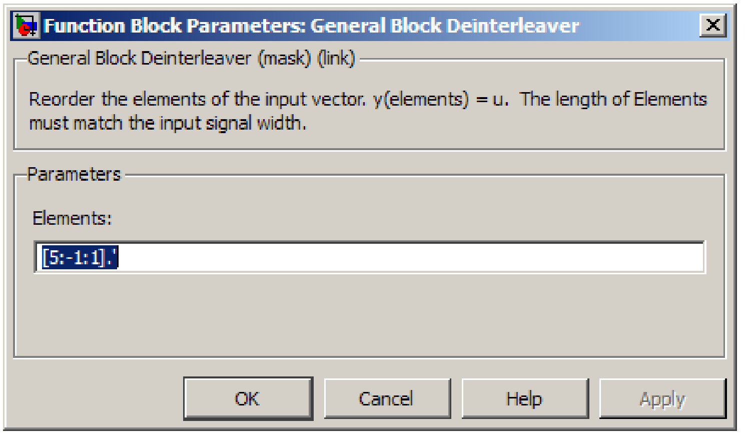
The **Elements** parameter must contain unique integers between 1 and N.

Both the input and the **Elements** parameter must be column vector signals.

This block accept the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

To use this block as an inverse of the General Block Interleaver block, use the same **Elements** parameter in both blocks. In that case, the two blocks are inverses in the sense that applying the General Block Interleaver block followed by the General Block Deinterleaver block leaves data unchanged.

Dialog Box



Elements

A vector of length N that lists the indices of the output elements that came from the input vector.

Examples

This example reverses the operation in the example on the General Block Interleaver block reference page. If you set **Elements** to `[4,1,3,2]'` and you set the General Block Deinterleaver block input to `[1;40;59;32]`, then the output of the General Block Deinterleaver block is `[40;32;59;1]`.

Pair Block

General Block Interleaver

See Also

`perms` (MATLAB function)

General Block Interleaver

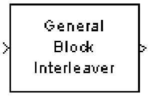
Purpose

Reorder symbols in input vector

Library

Block sublibrary of Interleaving

Description



The General Block Interleaver block rearranges the elements of its input vector without repeating or omitting any elements. If the input contains N elements, then the **Elements** parameter is a column vector of length N. The column vector indicates the indices, in order, of the input elements that form the length-N output vector; that is,

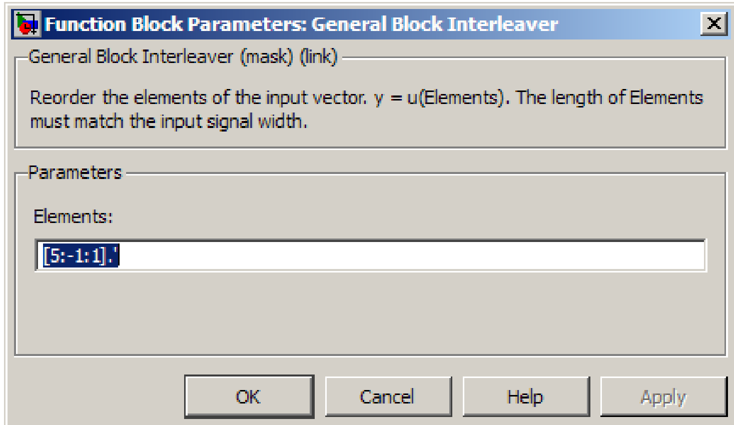
Output(k) = Input(**Elements**(k))

for each integer k between 1 and N. The contents of **Elements** must be integers between 1 and N, and must have no repetitions.

Both the input and the **Elements** parameter must be column vector signals.

This block accept the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

Dialog Box



Elements

A column vector of length N that lists the indices of the input elements that form the output vector.

Examples

If **Elements** is `[4;1;3;2]` and the input vector is `[40;32;59;1]`, then the output vector is `[1;40;59;32]`. Notice that all of these vectors have the same length and that the vector **Elements** is a permutation of the vector `[1:4]'`.

Pair Block

General Block Deinterleaver

See Also

`perms` (MATLAB function)

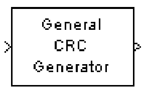
Purpose

Generate CRC bits according to generator polynomial and append to input data frames

Library

CRC sublibrary of Error Correction and Detection

Description



The General CRC Generator block generates cyclic redundancy code (CRC) bits for each input data frame and appends them to the frame. This block accepts a binary column vector input signal.

You specify the generator polynomial for the CRC algorithm using the **Generator polynomial** parameter. This block is general in the sense that the degree of the polynomial does not need to be a power of two. You represent the polynomial in one of these ways:

- As a binary row vector containing the coefficients in descending order of powers. For example, `[1 1 0 1]` represents the polynomial $x^3 + x^2 + 1$.
- As an integer row vector containing the powers of nonzero terms in the polynomial, in descending order. For example, `[3 2 0]` represents the polynomial $x^3 + x^2 + 1$.

You specify the initial state of the internal shift register by the **Initial states** parameter. The **Initial states** parameter is either a scalar or a binary row vector of length equal to the degree of the generator polynomial. A scalar value is expanded to a row vector of length equal to the degree of the generator polynomial. For example, the default initial state of `[0]` is expanded to a row vector of all zeros.

You specify the number of checksums that the block calculates for each input frame by the **Checksums per frame** parameter. The **Checksums per frame** value must evenly divide the size of the input frame. If the value of **Checksums per frame** is k, the block does the following:

1. Divides each input frame into k subframes of equal size
2. Prefixes the **Initial states** vector to each of the k subframes

3 Applies the CRC algorithm to each augmented subframe

4 Appends the resulting checksums at the end of each subframe

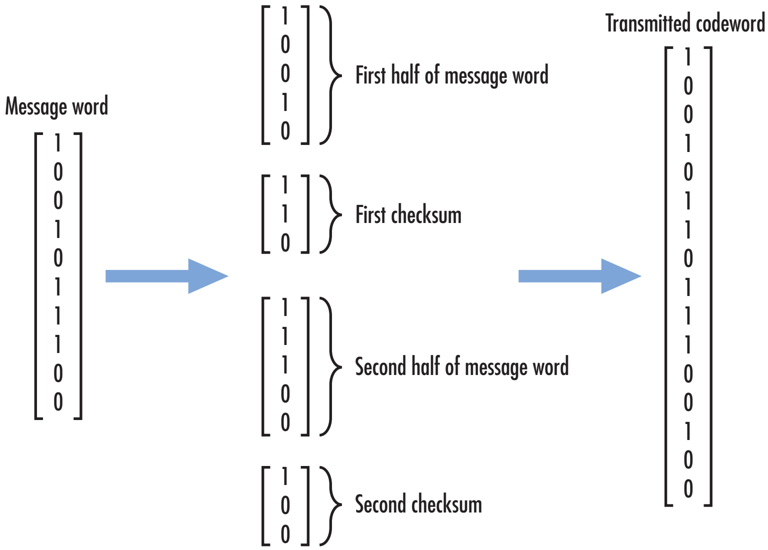
5 Outputs concatenated subframes

If the size of the input frame is m and the degree of the generator polynomial is r, the output frame has size $m + k * r$.

This block supports `double` and `boolean` data types. The block inherits the output data type from the input signal.

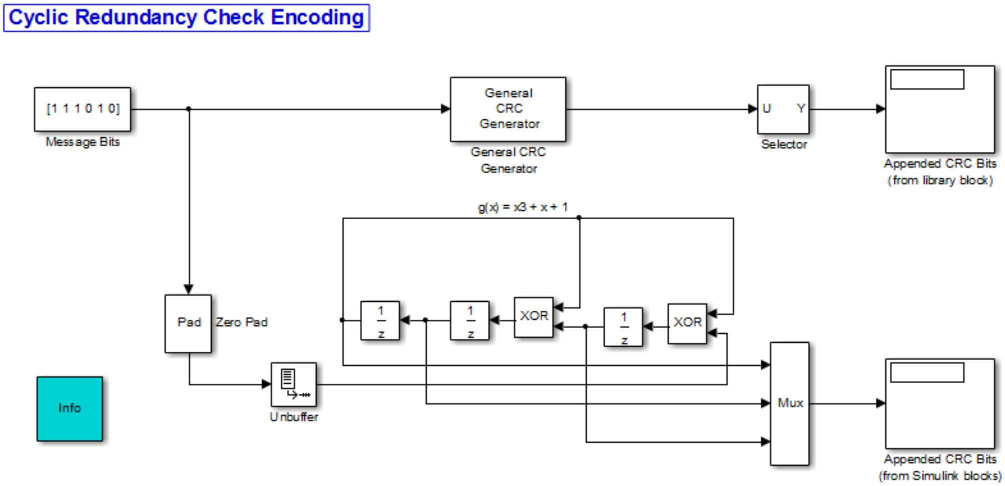
Example

Suppose the size of the input frame is 10, the degree of the generator polynomial is 3, **Initial states** is `[0]`, and **Checksums per frame** is `2`. The block divides each input frame into two subframes of size 5 and appends a checksum of size 3 to each subframe, as shown below. The initial states are not shown in this example, because an initial state of `[0]` does not affect the output of the CRC algorithm. The output frame then has size 5 + 3 + 5 + 3 = 16.



Example of Cyclic Redundancy Check Encoding

This example clarifies the operation of the General CRC Generator block by comparing the generated CRC bits from the library block with those generated from primitive Simulink blocks. To open the model, enter `doc_crcgen` at the MATLAB command line.



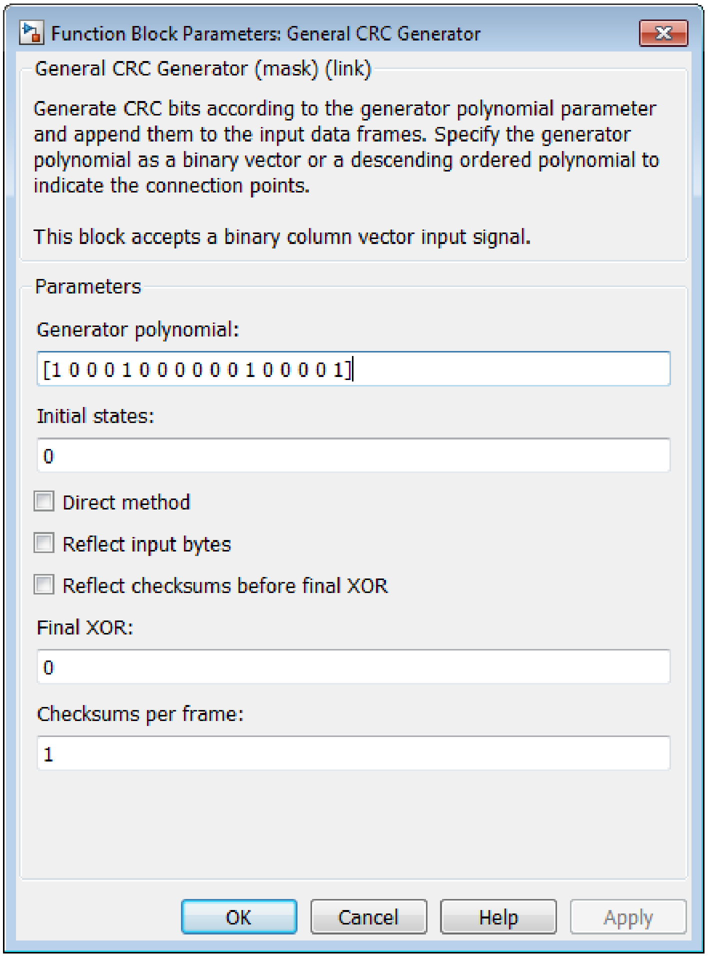
For a known input message with a length of 6 bits, the model generates CRC bits for a generator polynomial, $g(x) = x^3 + x + 1$, and a specific initial state of the register.

You can experiment with different initial states by changing the value of **Initial states** prior to running the simulation. For all values, the comparison (generated CRC bits from the library block with those generated from primitive Simulink blocks) holds true.

Using the General CRC Generator block allows you to easily specify the generator polynomial (especially for higher order polynomials).

Signal Attributes

The General CRC Generator block has one input port and one output port. Both ports support binary column vector signals.



Dialog Box

Generator polynomial

A binary or integer row vector specifying the generator polynomial, in descending order of powers.

Initial conditions
Binary scalar or a binary row vector of length equal to the degree of the generator polynomial, specifying the initial state of the internal shift register.

Direct method
When you select this check box, the object uses the direct algorithm for CRC checksum calculations. When you clear this check box, the object uses the non-direct algorithm for CRC checksum calculations.

Reflect input bytes
When you select this check box, the block flips the input data on a bytewise basis prior to entering the data into the shift register. For this application, the input frame length (and any current input frame length for variable-size signals) divided by the value for the **Checksums per frame** parameter must be a multiple of eight. When you clear this check box, the block does not flip the input data.

Reflect checksums before final XOR
When you select this check box, the block flips the CRC checksums around their centers after the input data are completely through the shift register. When you clear this check box, the block does not flip the CRC checksums.

Final XOR
Specify the value with which the CRC checksum is to be XORed as a binary scalar or vector. The block applies the XOR operation just prior to appending the input data. The vector length is the degree of the generator polynomial that you specify in the **Generator polynomial** parameter. When you specify the final XOR value as a scalar, the block expands the value to a row vector with a length equal to the degree of the generator polynomial. The default value of this parameter is `0`, which is equivalent to no XOR operation.

Checksums per frame
Specify the number of checksums the block calculates for each input frame. This value must be a positive integer. The input

frame length (and any current input frame length for variable-size signals) must be a multiple of this parameter value.

Algorithm

For a description of the CRC algorithm as implemented by this block, see "Cyclic Redundancy Check Codes" in *Communications System Toolbox User's Guide*.

References

[1] Sklar, Bernard, *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, N.J., Prentice-Hall, 1988.

[2] Wicker, Stephen B., *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, N.J., Prentice Hall, 1995.

Pair Block

General CRC Syndrome Detector

See Also

CRC-N Generator, CRC-N Syndrome Detector

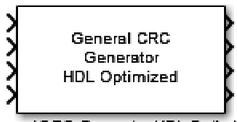
Purpose

Generate CRC code bits and append to input data, optimized for HDL code generation

Library

CRC sublibrary of Error Correction and Detection

Description

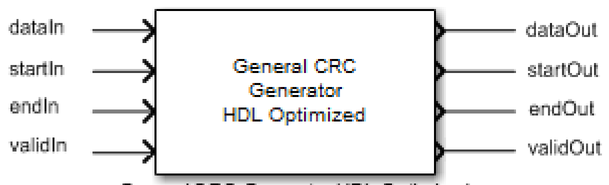


General CRC Generator HDL Optimized

This hardware-friendly CRC generator block, like the General CRC Generator block, generates the CRC bits and appends them to the input message bits. The output consists of CRC checksum plus the message. With the General CRC Generator HDL Optimized block, the processing is optimized for HDL code generation. Instead of processing an entire frame at once, the block processes samples of data. Control signals are added at both input and output for easy data synchronization.

Signal Attributes

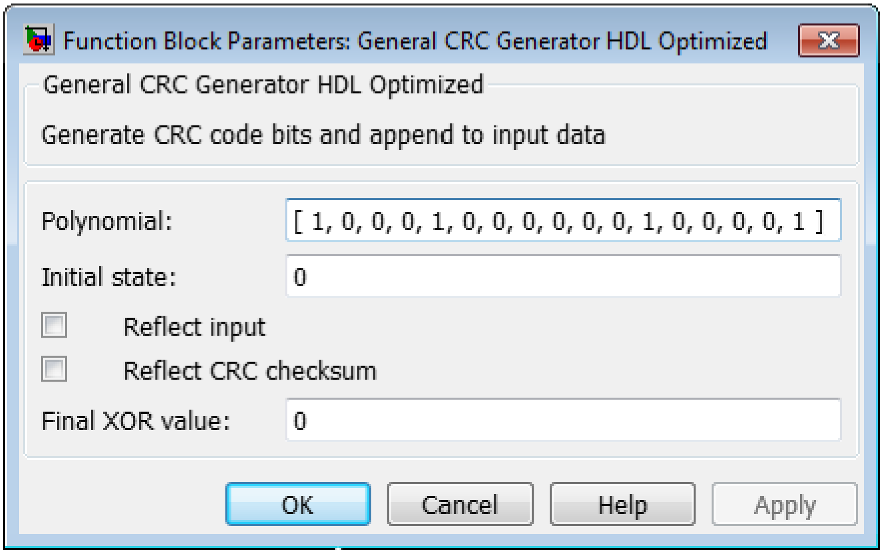
The General CRC Generator HDL Optimized block has four input ports and four output ports.



General CRC Generator HDL Optimized

Port	Direction	Description	Data Type
dataIn	Input	Message data. Data width is less than or equal to the CRC length, and the CRC length should be divisible by the data width. For example, for CRC-CCITT/CRC-16, the valid data widths are 16, 8, 4, 2 and 1.	Column vector of double, boolean, or ufix1
startIn	Input	Indicates the start of a frame of data.	Boolean or ufix1
endIn	Input	Indicates the end of a frame of data.	Boolean or ufix1
validIn	Input	Indicates that input data is valid.	Boolean or ufix1
dataOut	Output	Message data with the checksum appended. The data width is the same as the input data port.	Column vector of double, boolean, or ufix1
startOut	Output	Indicates the start of a frame of data.	Boolean or ufix1
endOut	Output	Indicates the end of a frame of data, including checksum.	Boolean or ufix1
validOut	Output	Indicates that output data is valid.	Boolean or ufix1

Dialog Box



Polynomial

A double, boolean, or ufix1 row or column vector specifying the polynomial, in descending order of powers. CRC length is length(polynomial)-1. The default value is [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1].

Initial state

A double, boolean, or ufix1 scalar or vector of length equal to the CRC length, specifying the initial state of the internal shift register. The default value is 0.

Reflect input

- The input data width must be a multiple of 8.
- When checked, each input byte is flipped before entering the shift register.
- When unchecked, the message data is passed to the shift register unchanged.

The default value is unchecked.

Reflect CRC checksum

- The CRC length must be a multiple of 8.
- When checked, each checksum byte is flipped before it is passed to the final XOR stage.
- When unchecked, the checksum byte is passed to the final XOR stage unchanged.

The default value is unchecked.

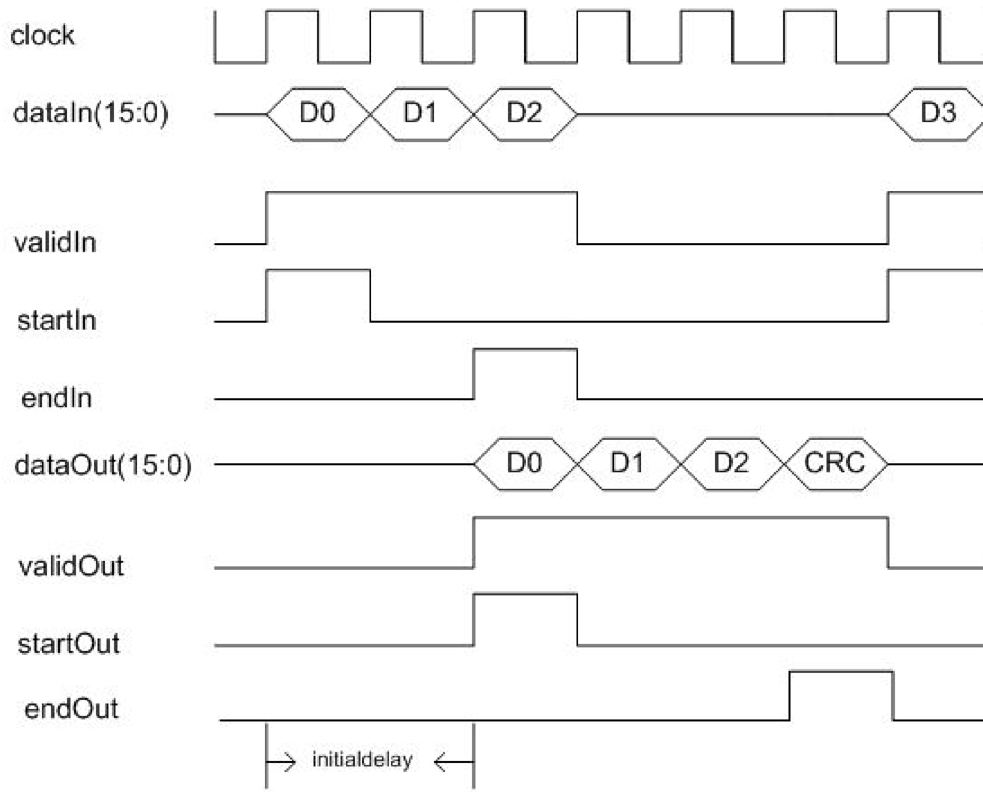
Final XOR value

The value with which the CRC checksum is to be XORed just prior to being appended to the input data. A double, boolean, or ufix1 scalar or vector of length equal to the CRC length, specifying the FinalXOR value. The default value is 0.

Algorithm

Timing Diagram

Timing diagram of CRC generator



Initial Delay

The General CRC Generator HDL Optimized block introduces a latency on the output. This latency can be computed with the following equation:

`initialdelay = CRC length/input data width + 2`

Example

See Using HDL Optimized CRC Library Blocks.

Pair Block General CRC Syndrome Detector HDL Optimized

See Also General CRC Generator

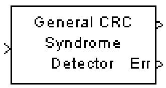
Purpose

Detect errors in input data frames according to generator polynomial

Library

CRC sublibrary of Error Correction and Detection

Description



The General CRC Syndrome Detector block computes checksums for its entire input frame. This block accepts a binary column vector input signal.

The block's second output is a vector whose size is the number of checksums, and whose entries are 0 if the checksum computation yields a zero value, and 1 otherwise. The block's first output is the set of message words with the checksums removed.

The first output is the data frame with the CRC bits removed and the second output indicates if an error was detected in the data frame.

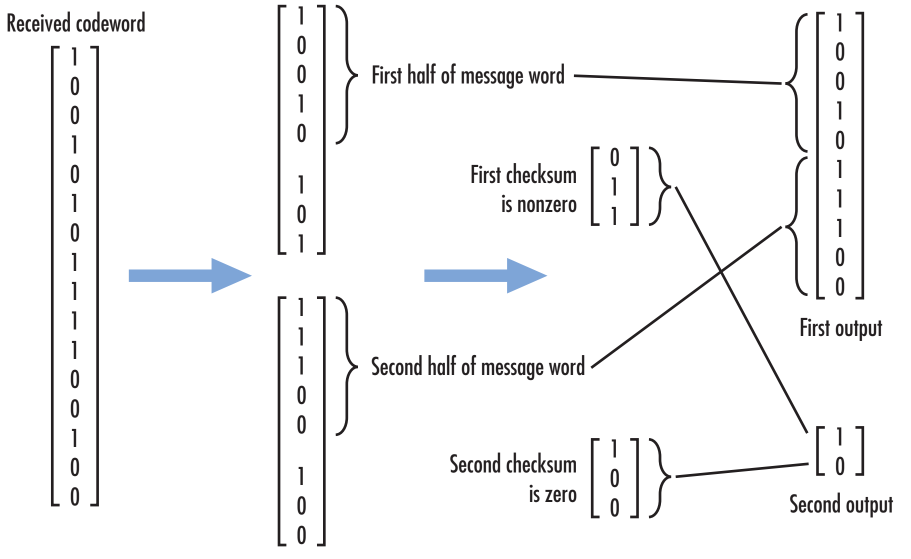
The block's parameter settings should agree with those in the General CRC Generator block.

You specify the number of checksums the block calculates for each frame by the **Checksums per frame** parameter. If the **Checksums per frame** value is k, the size of the input frame is n, and the degree of the generator polynomial is r, then k must divide n - k*r, which is the size of the message word.

This block supports `double` and `boolean` data types. The block inherits the output data type from the input signal.

Example

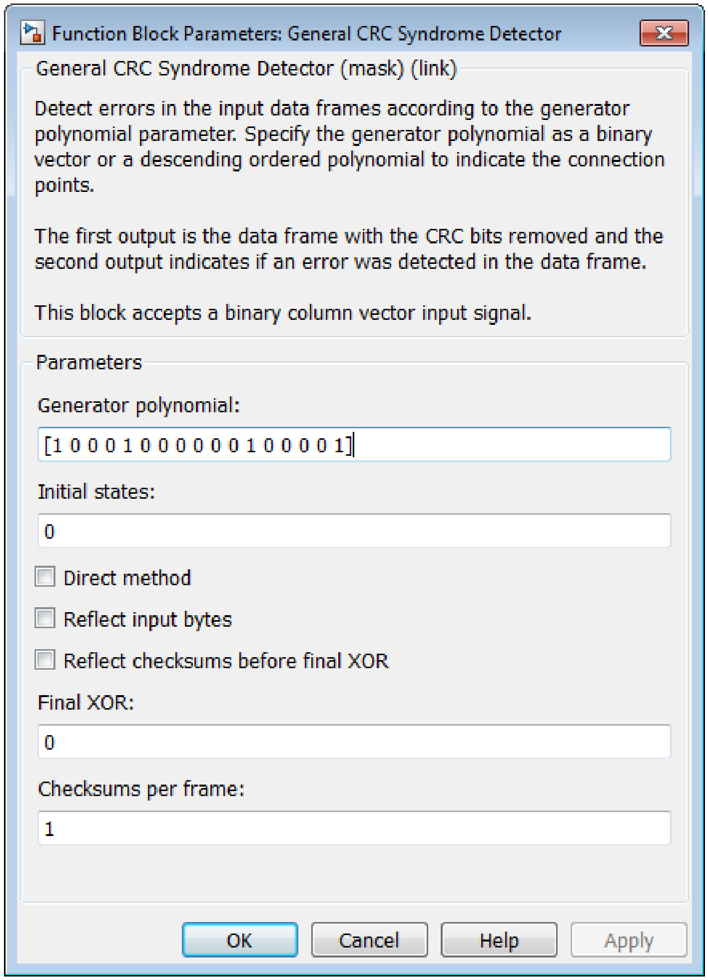
Suppose the received codeword has size 16, the generator polynomial has degree 3, **Initial states** is `[0]`, and **Checksums per frame** is `2`. The block computes the two checksums of size 3, one from the first half of the received codeword, and the other from the second half of the received codeword, as shown in the following figure. The initial states are not shown in this example, because an initial state of `[0]` does not affect the output of the CRC algorithm. The block concatenates the two halves of the message word as a single vector of size 10 and outputs this vector through the first output port. The block outputs a 2-by-1 binary frame vector whose entries depend on whether the computed checksums are zero. The following figure shows an example in which the first checksum is nonzero and the second checksum is zero. This indicates that an error occurred in transmitting the first half of the codeword.



Signal Attributes

The General CRC Syndrome Detector block has one input port and two output ports. These ports accept binary column vector signals.

Dialog Box



Generator polynomial

A binary or integer row vector specifying the generator polynomial, in descending order of powers.

Initial conditions
A binary scalar or a binary row vector of length equal to the degree of the generator polynomial, specifying the initial state of the internal shift register.

Direct method
When you select this check box, the object uses the direct algorithm for CRC checksum calculations. When you clear this check box, the object uses the non-direct algorithm for CRC checksum calculations.

Reflect input bytes
When you select this check box, the block flips the input data on a bytewise basis prior to entering the data into the shift register. For this application, the input frame length (and any current input frame length for variable-size signals) divided by the value for the **Checksums per frame** parameter minus the degree of the generator polynomial, which you specify in the **Generator polynomial** parameter, must be a multiple of eight. When you clear this check box, the block does not flip the input data.

Reflect checksums before final XOR
When you select this check box, the block flips the CRC checksums around their centers after the input data are completely through the shift register. When you clear this check box, the block does not flip the CRC checksums.

Final XOR
Specify the value with which the CRC checksum is to be XORed as a binary scalar or vector. The block applies the XOR operation just prior to appending the input data. The vector length is the degree of the generator polynomial that you specify in the **Generator polynomial** parameter. When you specify the final XOR value as a scalar, the block expands the value to a row vector with a length equal to the degree of the generator polynomial. The default value of this parameter is `0`, which is equivalent to no XOR operation.

Checksums per frame

Specify the number of checksums the block calculates for each input frame. This value must be a positive integer. The input frame length (and any current input frame length for variable-size signals) must be a multiple of this parameter value.

Algorithm

For a description of the CRC algorithm as implemented by this block, see "Cyclic Redundancy Check Codes" in *Communications System Toolbox User's Guide*.

References

[1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, N.J., Prentice-Hall, 1988.

[2] Wicker, Stephen B., *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, N.J., Prentice Hall, 1995.

Pair Block

General CRC Generator

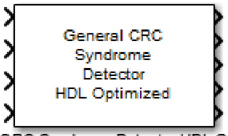
See Also

CRC-N Generator, CRC-N Syndrome Detector

General CRC Syndrome Detector HDL Optimized

Purpose Detect errors in input data using CRC

Library CRC sublibrary of Error Correction and Detection



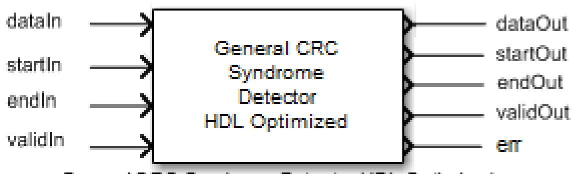
General CRC Syndrome Detector HDL Optimized

Description

This hardware-friendly CRC detector block performs a CRC on data and compares the resulting checksum with the appended checksum. An error is detected if the two checksums do not match. Instead of frame processing, the block processes data at the streaming mode. Control signals are added at both input and output of the block for easy data synchronization.

Signal Attributes

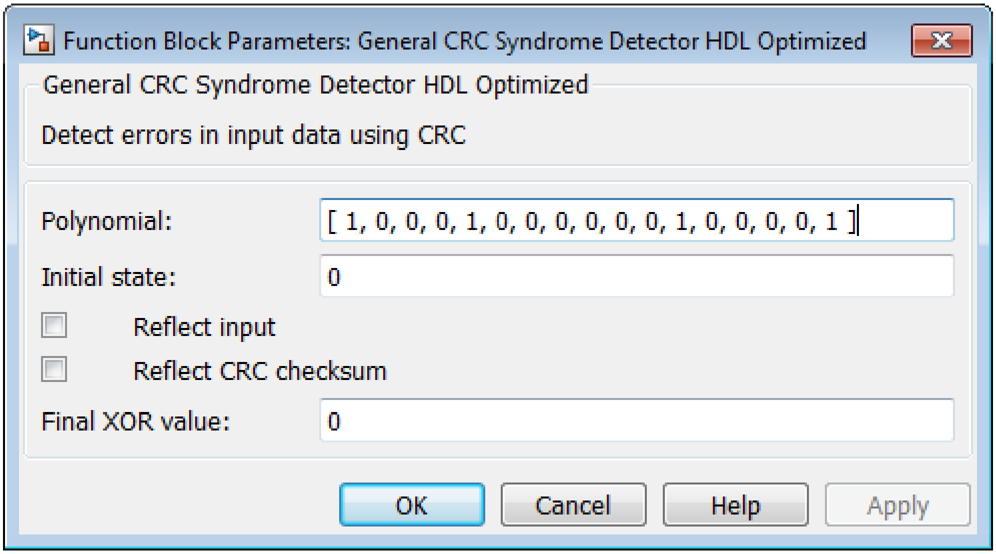
The General CRC Syndrome Detector HDL Optimized block has four input ports and five output ports.



General CRC Syndrome Detector HDL Optimized

Port	Direction	Description	Data Type
dataIn	Input	Message data. Data width is less than or equal to the CRC length, and the CRC length should be divisible by the data width. For example, for CRC-CCITT/CRC-16, the valid data widths are 16, 8, 4, 2 and 1.	Column vector of double, Boolean, or ufix1
startIn	Input	Indicates the start of a frame of data.	Boolean or ufix1
endIn	Input	Indicates the end of a frame of data.	Boolean or ufix1
validIn	Input	Indicates that input data is valid.	Boolean or ufix1
dataOut	Output	Message data with the checksum appended. The data width is the same as the input data port.	Column vector of double, Boolean, or ufix1
startOut	Output	Indicates the start of a frame of data.	Boolean or ufix1
endOut	Output	Indicates the end of a frame of data, including checksum.	Boolean or ufix1
validOut	Output	Indicates that output data is valid.	Boolean or ufix1
err	Output	Indicates the corruption of the received data when error is high.	Boolean or ufix1

Dialog Box



Polynomial

A double, boolean, or ufix1 row or column vector specifying the polynomial, in descending order of powers. CRC length is length(polynomial)-1. The default value is [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1].

Initial state

A double, boolean, or ufix1 scalar or vector of length equal to the CRC length, specifying the initial state of the internal shift register. The default value is 0.

Reflect input

- The input data width must be a multiple of 8.
- When checked, each input byte is flipped before entering the shift register.
- When unchecked, the message data is passed to the shift register unchanged.

The default value is unchecked.

Reflect CRC checksum

- The CRC length must be a multiple of 8.
- When checked, each checksum byte is flipped before it is passed to the final XOR stage.
- When unchecked, the checksum byte is passed to the final XOR stage unchanged.

The default value is unchecked.

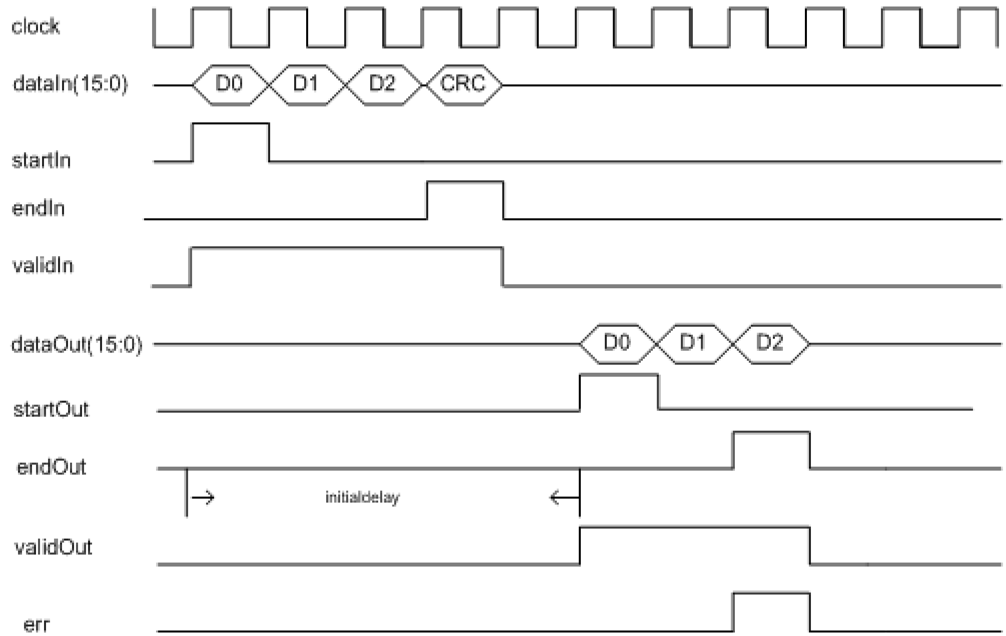
Final XOR value

The value with which the CRC checksum is to be XORed just prior to being appended to the input data. A double, boolean, or ufix1 scalar or vector of length equal to the CRC length, specifying the FinalXOR value. The default value is 0.

Algorithm

Timing Diagram

Timing diagram of CRC detector



Initial Delay

The General CRC Syndrome Detector HDL Optimized block introduces a latency on the output. This latency can be computed with the following equation:

```
initialdelay = (3 * CRC length/input data width) + 2
```

Example

See Using HDL Optimized CRC Library Blocks.

Pair Block

General CRC Generator HDL Optimized

See Also

General CRC Syndrome Detector

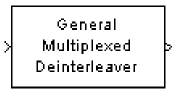
Purpose

Restore ordering of symbols using specified-delay shift registers

Library

Convolutional sublibrary of Interleaving

Description

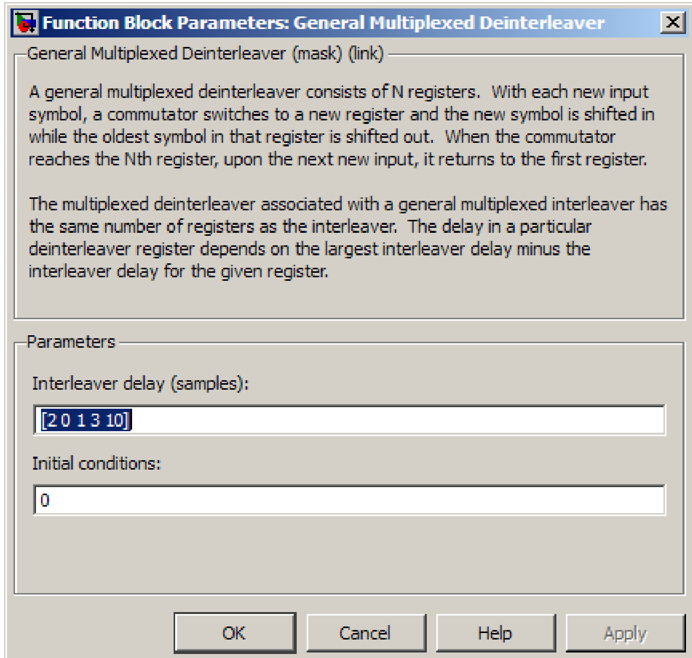


The General Multiplexed Deinterleaver block restores the original ordering of a sequence that was interleaved using the General Multiplexed Interleaver block.

In typical usage, the parameters in the two blocks have the same values. As a result, the **Interleaver delay** parameter, *V*, specifies the delays for each shift register in the corresponding *interleaver*, so that the delays of the deinterleaver's shift registers are actually `max(V)-V`.

This block accepts a scalar or column vector input signal, which can be real or complex. The output signal has the same sample time as the input signal.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The block inherist data type of this output will be the same as that of the input signal.



Dialog Box

Interleaver delay (samples)

A vector that lists the number of symbols that fit in each shift register of the corresponding interleaver. The length of this vector is the number of shift registers.

Initial conditions

The values that fill each shift register when the simulation begins.

Pair Block

General Multiplexed Interleaver

See Also

Convolutional Deinterleaver, Helical Deinterleaver

References

[1] Heegard, Chris and Stephen B. Wicker. *Turbo Coding*. Boston: Kluwer Academic Publishers, 1999.

Purpose

Permute input symbols using set of shift registers with specified delays

Library

Convolutional sublibrary of Interleaving

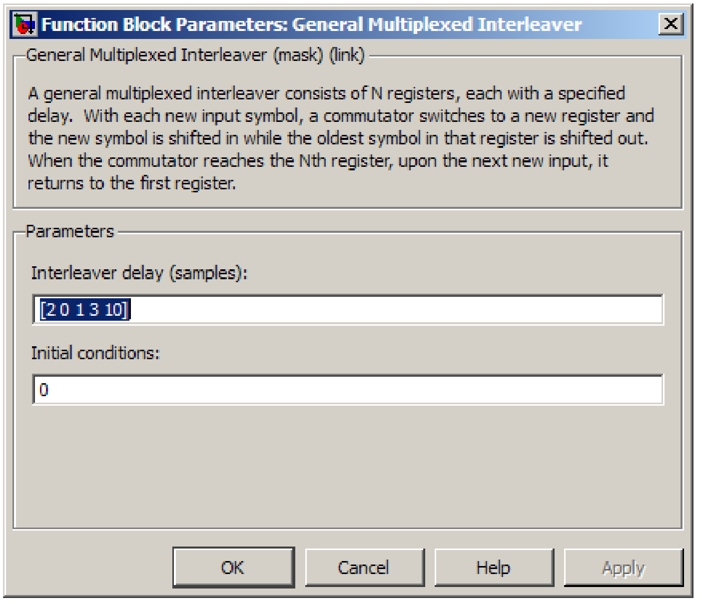
Description

The General Multiplexed Interleaver block permutes the symbols in the input signal. Internally, it uses a set of shift registers, each with its own delay value.

This block accepts a scalar or column vector input signal, which can be real or complex. The input and output signals have the same sample time.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal has the same data type as the input signal.

General Multiplexed Interleaver



Dialog Box

Interleaver delay (samples)

A column vector listing the number of symbols that fit into each shift register. The length of this vector is the number of shift registers. (In sample-based mode, it can also be a row vector.)

Initial conditions

The values that fill each shift register at the beginning of the simulation.

If **Initial conditions** is a scalar, then its value fills all shift registers. If **Initial conditions** is a column vector, then each entry fills the corresponding shift register. (In sample-based mode, **Initial conditions** can also be a row vector.) If a given shift register has zero delay, then the value of the corresponding entry in the **Initial conditions** vector is unimportant.

Pair Block

General Multiplexed Deinterleaver

See Also

Convolutional Interleaver, Helical Interleaver

References

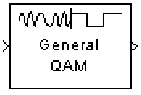
[1] Heegard, Chris and Stephen B. Wicker. *Turbo Coding*. Boston: Kluwer Academic Publishers, 1999.

General QAM Demodulator Baseband

Purpose Demodulate QAM-modulated data

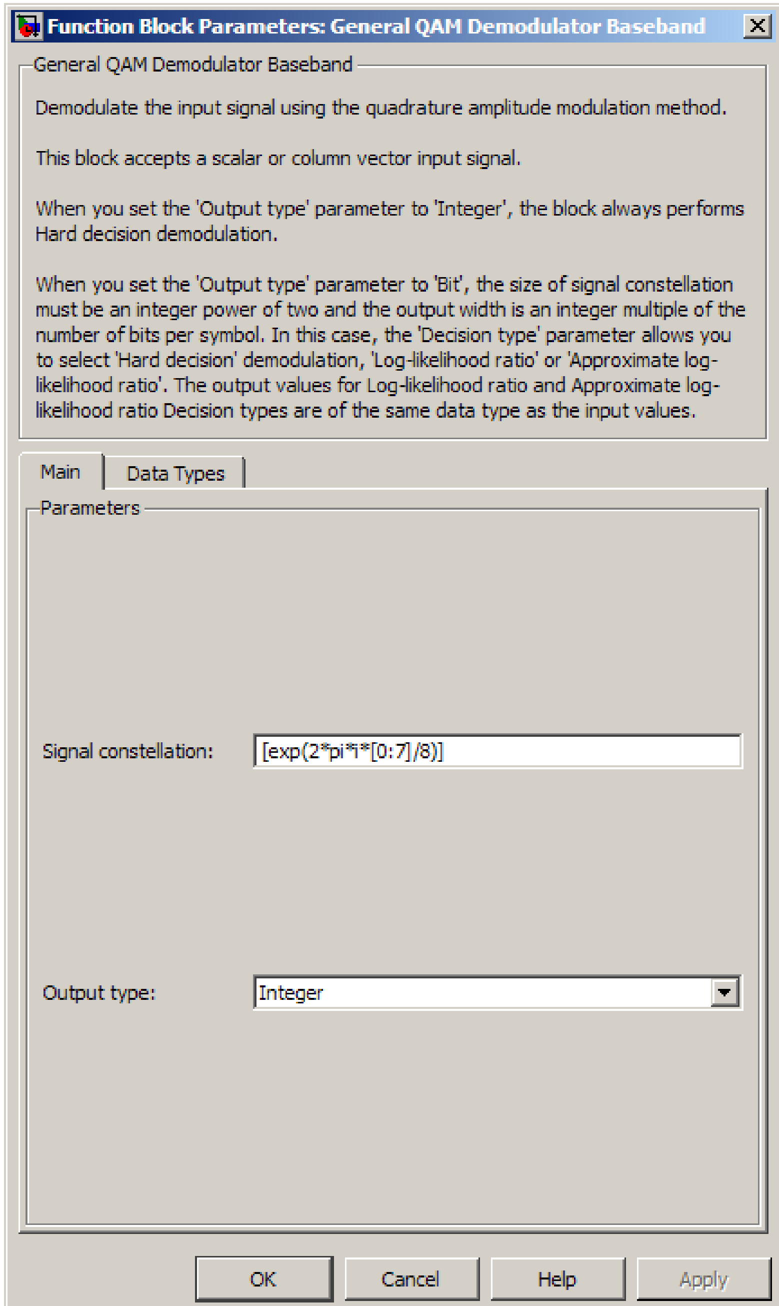
Library AM, in Digital Baseband sublibrary of Modulation

Description The General QAM Demodulator Baseband block demodulates a signal that was modulated using quadrature amplitude modulation. The input is a baseband representation of the modulated signal.



The input must be a discrete-time complex signal. The **Signal constellation** parameter defines the constellation by listing its points in a length-M vector of complex numbers. The block maps the *m*th point in the **Signal constellation** vector to the integer *m*-1.

This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see the "Supported Data Types" on page 2-348 table on this page.



Dialog Box

Signal constellation

A real or complex vector that lists the constellation points.

Output type

Determines whether the block produces integers or binary representations of integers.

If you set this parameter to `Integer`, the block produces integers.

If you set this parameter to `Bit`, the block produces a group of K bits, called a *binary word*, for each symbol, when **Decision type** is set to `Hard decision`. If **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the block outputs bitwise LLR and approximate LLR, respectively.

Decision type

This field appears when `Bit` is selected in the pull-down list **Output type**.

Specifies the use of hard decision, LLR, or approximate LLR during demodulation. See "Exact LLR Algorithm" and "Approximate LLR Algorithm" in the *Communications System Toolbox User's Guide* for algorithm details.

Noise variance source

This field appears when you set `Approximate log-likelihood ratio` or `Log-likelihood ratio` for **Decision type**.

When you set this parameter to `Dialog`, you can then specify the noise variance in the **Noise variance** field. When you set this option to `Port`, a port appears on the block through which the noise variance can be input.

Noise variance

This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

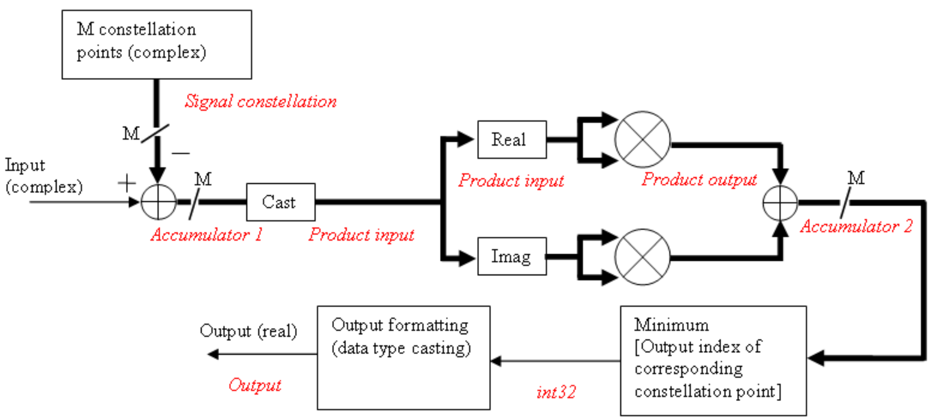
If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

The LLR algorithm involves computing exponentials of very large or very small numbers using finite precision arithmetic and would yield:

- `Inf` to `-Inf` if **Noise variance** is very high
- `NaN` if **Noise variance** and signal power are both very small

In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.

Fixed-Point Signal Flow Diagrams

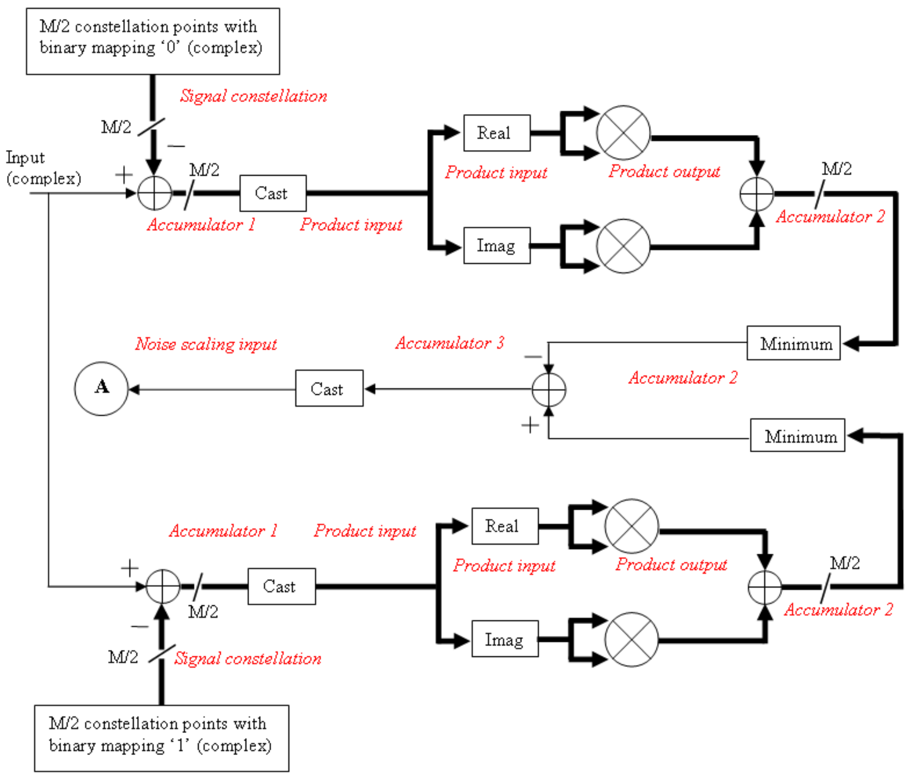


Fixed-Point Signal Flow Diagram for Hard Decision Mode

Note In the figure above, *M* represents the size of the **Signal constellation** .

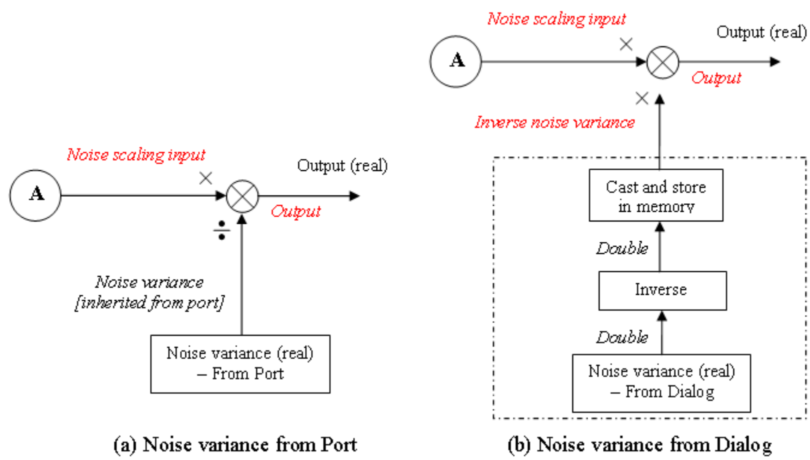
The general QAM Demodulator Baseband block supports fixed-point operations for computing Hard Decision (**Output** type set to `Bit` and **Decision type** is set to `Hard decision`) and Approximate LLR (**Output type** is set to `Bit` and **Decision type** is set to `Approximate Log-Likelihood ratio`) output values. The input values must have fixed-point data type for fixed-point operations.

Note Fixed-Point operations are NOT yet supported for Exact LLR output values.



Fixed-Point Signal Flow Diagram for Approximate LLR Mode

Note In the figure above, M represents the size of the **Signal constellation**.

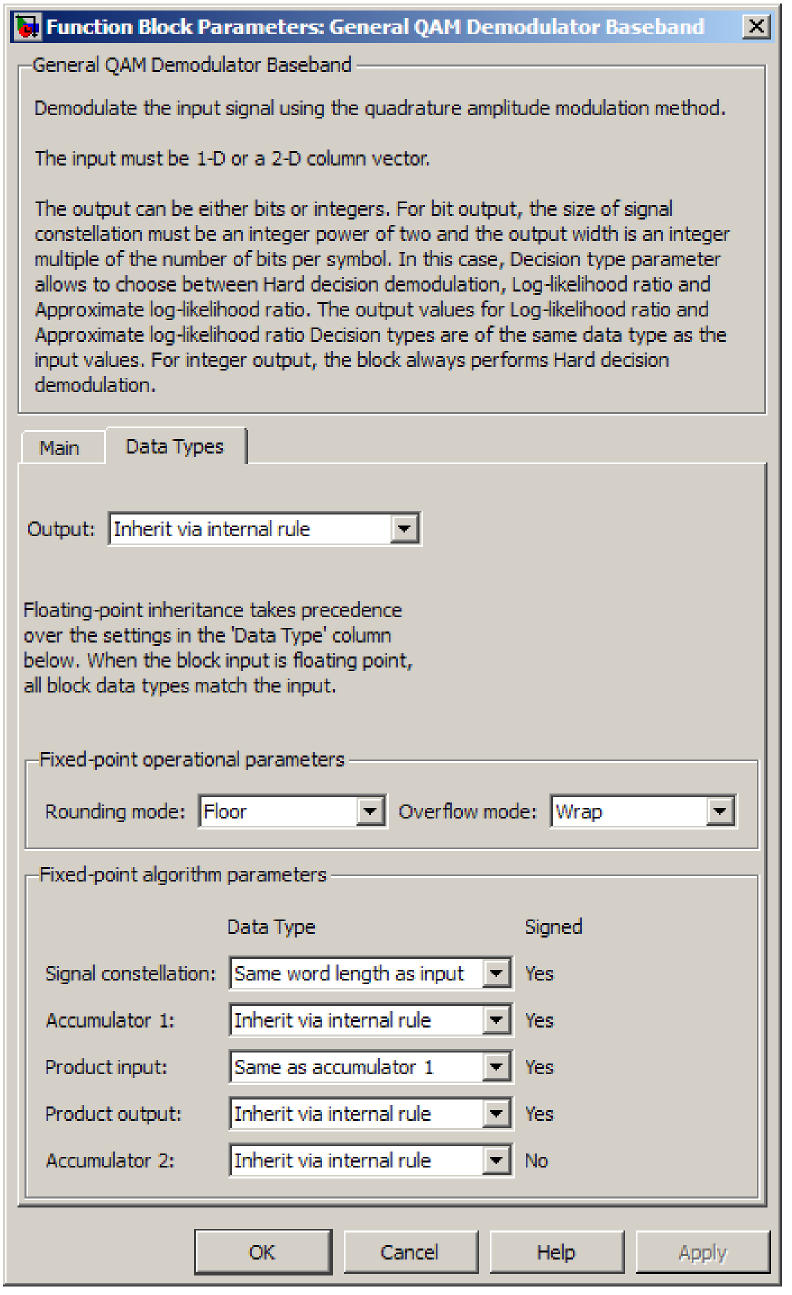


Fixed-Point Signal Flow Diagram for Approximate LLR Mode: Noise Variance Operation Modes

---

Note If **Noise variance** is set to `Dialog`, the block performs the operations shown inside the dotted line once during initialization. The block also performs these operations if the **Noise variance** value changes during simulation.

Data Types Attributes



Fixed-Point Attributes for Hard Decision Mode

Output

The block supports the following Output options:

When you set the parameter to `Inherit via internal rule` (default setting), the block inherits the output data type from the input port. The output data type is the same as the input data type if the input is of type `single` or `double`.

For integer outputs, you can set this block's output to `Inherit via internal rule` (default setting), `Smallest unsigned integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`.

For bit outputs, when you set **Decision type** to `Hard decision`, you can set the output to `Inherit via internal rule`, `Smallest unsigned integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, or `double`.

When you set **Decision type** to `Hard decision` or `Approximate log-likelihood ratio` and the input is a floating point data type, then the output inherits its data type from the input. For example, if the input is of data type `double`, the output is also of data type `double`. When you set **Decision type** to `Hard decision` or `Approximate log-likelihood ratio`, and the input is a fixed-point signal, the **Output** parameter, located in the Fixed-Point algorithm parameters region of the Data-Type tab, specifies the output data type.

When you set the parameter to `Smallest unsigned integer`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box. If you select `ASIC/FPGA` in the **Hardware Implementation** pane, the output data type is the ideal minimum size, i.e., `ufix(1)` for bit outputs, and $ufix(\lceil \log_2 M \rceil)$ for integer outputs. For all other choices, the **Output** data type is an unsigned integer with the smallest available word length large enough to fit the ideal minimum size, usually corresponding to the size of a char (e.g., `uint8`).

Rounding Mode Parameter

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

For more information, see “Rounding Modes” in the DSP System Toolbox documentation “Rounding Mode: Simplest” in the Fixed-Point Designer documentation.

Overflow Mode Parameter

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

- `Saturate` represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used.
- `Wrap` uses modulo arithmetic to cast an overflow back into the representable range of the data type. See Modulo Arithmetic for more information.

For more information, see the Rounding Mode Parameter subsection of “Specify Fixed-Point Attributes for Blocks”.

Signal constellation

Use this parameter to define the data type of the **Signal constellation** parameter.

- When you select `Same word length as input` the word length of the **Signal constellation** parameter matches that of the input to the block. The fraction length is computed to provide the best precision for given signal constellation values.
- When you select `Specify word length`, the **Word Length** field appears, and you may enter a value for the word length.

The fraction length is computed to provide the best precision for given signal constellation values.

Accumulator 1

Use this parameter to specify the data type for **Accumulator 1**:

- When you select `Inherit via internal rule`, the block automatically calculates the output word and fraction lengths. For more information, see the "Inherit via Internal Rule" subsection of the *DSP System Toolbox User's Guide*.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of **Accumulator 1**, in bits.

Product Input

Use this parameter to specify the data type for **Product input**.

- When you select `Same as accumulator 1`, the **Product Input** characteristics match those of **Accumulator 1**.
- When you select `Binary point scaling` you can enter the word length and the fraction length of **Product input**, in bits.

Product Output

Use this parameter to select the data type for Product output.

- When you select `Inherit via internal rule`, the block automatically calculates the output signal type. For more information, see the `Inherit via Internal Rule` subsection of the *DSP System Toolbox User's Guide*.
- When you select `Binary point scaling` enter the word length and the fraction length for **Product output**, in bits.

Accumulator 2

Use this parameter to specify the data type for **Accumulator 2**:

- When you select `Inherit via internal rule`, the block automatically calculates the accumulator data type. The internal rule calculates the ideal, full-precision word length and fraction length as follows:

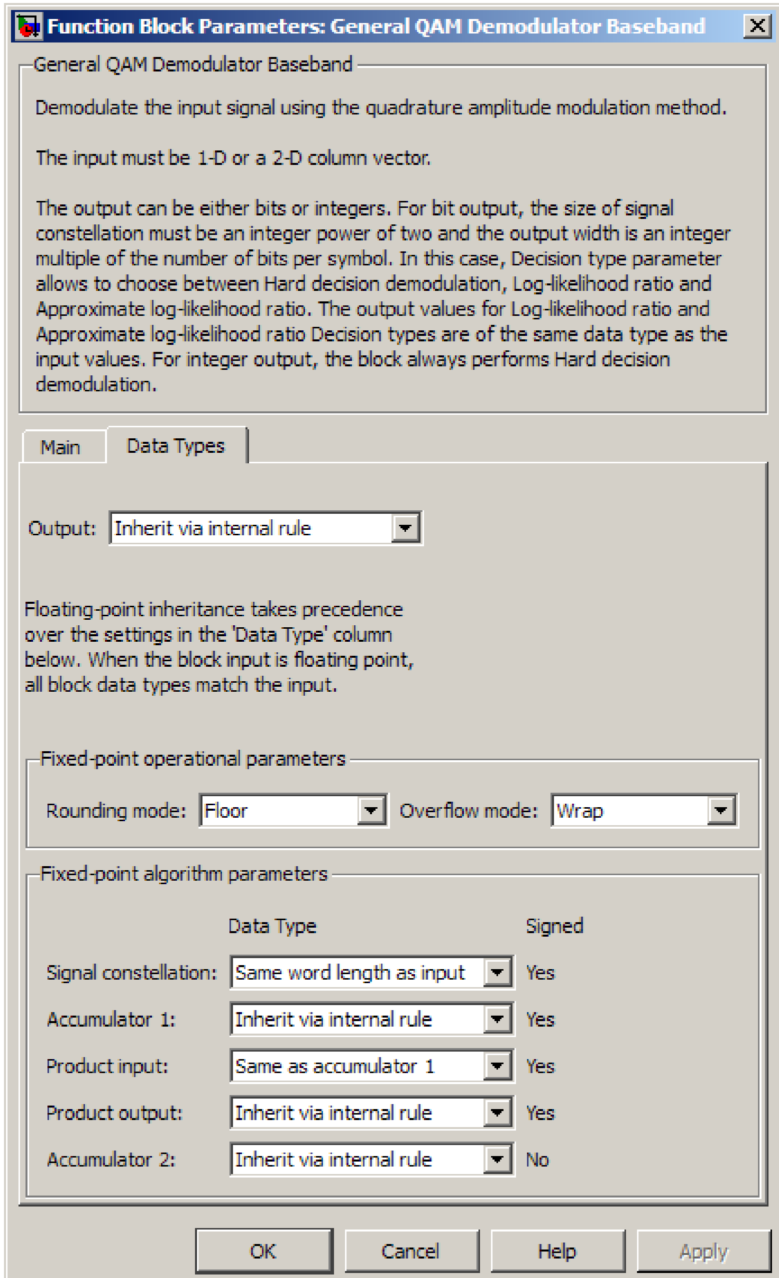
  $$WL_{\text{ideal accumulator 2}} = WL_{\text{input to accumulator 2}}$$

$FL_{\text{ideal accumulator 2}} = FL_{\text{ input to accumulator 2}}$

After the full-precision result is calculated, your particular hardware may still affect the final word and fraction lengths set by the internal rule. For more information, see The Effect of the Hardware Implementation Pane on the Internal Rule subsection of the *DSP System Toolbox User's Guide*.

The internal rule always sets the sign of data-type to `Unsigned` .

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of **Accumulator 2**, in bits.



Fixed-Point Attributes for Approximate LLR Mode

The settings for the following fixed-point parameters only apply when you set **Decision type** to `Approximate log-likelihood ratio`.

Accumulator 3

When you select `Inherit via internal rule`, the block automatically calculates the accumulator data type. The internal rule first calculates ideal, full-precision word length and fraction length as follows:

$$WL_{\text{ideal accumulator 3}} = WL_{\text{input to accumulator 3}} + 1$$

$$FL_{\text{ideal accumulator 3}} = FL_{\text{input to accumulator 3}}.$$

After the full-precision result is calculated, your particular hardware may still affect the final word and fraction lengths set by the internal rule. For more information, see The Effect of the Hardware Implementation Pane on the Internal Rule subsection of the *DSP System Toolbox User's Guide*.

The internal rule always sets the sign of data-type to `Signed`.

Noise scaling input

- When you select `Same as accumulator 3`, the **Noise scaling input** characteristics match those of **Accumulator 3**.
- When you select `Binary point scaling` you are able to enter the word length and the fraction length of **Noise scaling input**, in bits.

Inverse noise variance

This field appears when **Noise variance** source is set to Dialog.

- When you select `Same word length as input` the word length of the **Inverse noise variance** parameter matches that of the input to the block. The fraction length is computed to provide the best precision for a given inverse noise variance value.
- When you select `Specify word length`, the **Word Length** field appears, and you may enter a value for the word length.

The fraction length is computed to provide the best precision for a given inverse noise variance value.

Output

When you select `Inherit via internal rule`, the **Output data type** is automatically set for you.

If you set the **Noise variance source** parameter to `Dialog`, the output is a result of product operation as shown in the Noise Variance Operation Modes Signal Flow Diagram Fixed-Point Signal Flow Diagram for Approximate LLR Mode: Noise Variance Operation Modes on page 2-339. In this case, it follows the internal rule for Product data types specified in the Inherit via Internal Rule subsection of the *DSP System Toolbox User's Guide*.

If the **Noise variance source** parameter is set to `Port`, the output is a result of division operation as shown in the signal flow diagram. In this case, the internal rule calculates the ideal, full-precision word length and fraction length as follows:

$$WL_{\text{output}} = \max(WL_{\text{Noise scaling input}}, WL_{\text{Noise variance}})$$

$$FL_{\text{output}} = FL_{\text{Noise scaling input (dividend)}} - FL_{\text{Noise variance (divisor)}} .$$

After the full-precision result is calculated, your particular hardware may still affect the final word and fraction lengths set by the internal rule. For more information, see "The Effect of the Hardware Implementation Pane on the Internal Rule" subsection of the *DSP System Toolbox User's Guide*.

The internal rule for **Output** always sets the sign of data-type to `Signed`.

For additional information about the parameters pertaining to fixed-point applications, see "Specify Fixed-Point Attributes for Blocks".

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed–point when **Output type** is `Integer` or **Output type** is `Bit` and **Decision type** is either `Hard-decision` or `Approximate LLR`
Var	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point • Boolean when **Output type** is `Bit` and **Decision type** is `Hard-decision`. • 8-, 16-, and 32-bit signed integers when **Output type** is `Integer` or **Output type** is `Bit` and **Decision type** is `Hard-decision` • 8-, 16-, and 32-bit unsigned integers when **Output type** is `Integer` or **Output type** is `Bit` and **Decision type** is `Hard-decision` • ufix(1) in ASIC/FPGA when **Output type** is `Bit` and **Decision type** is `Hard-decision` • $ufix(\lceil \log_2 M \rceil)$ in ASIC/FPGA when **Output type** is `Integer` • Signed fixed-point when **Output type** is `Bit` and **Decision type** is `Approximate LLR`

Pair Block

General QAM Modulator Baseband

See Also

Rectangular QAM Demodulator Baseband

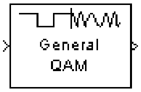
Purpose

Modulate using quadrature amplitude modulation

Library

AM, in Digital Baseband sublibrary of Modulation

Description



The General QAM Modulator Baseband block modulates using quadrature amplitude modulation. The output is a baseband representation of the modulated signal.

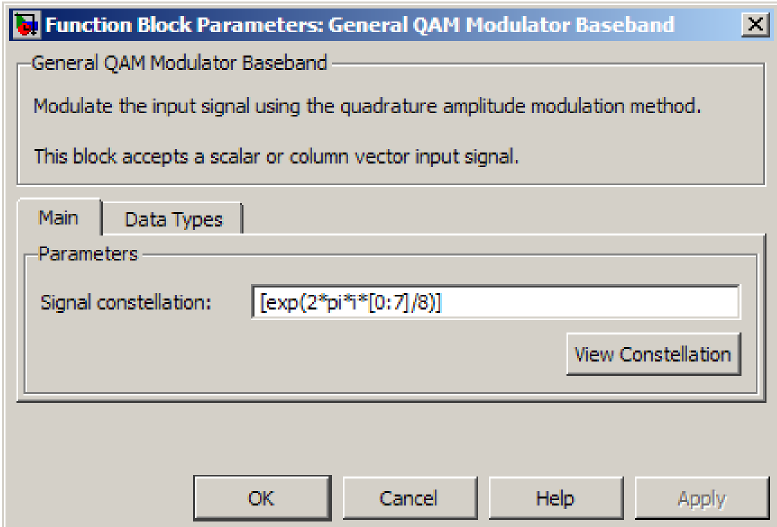
The **Signal constellation** parameter defines the constellation by listing its points in a length-M vector of complex numbers. The input signal values must be integers between 0 and M-1. The block maps an input integer m to the (m+1)st value in the **Signal constellation** vector.

This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see the “Supported Data Types” on page 2-351 table on this page.

Constellation Visualization

The General QAM Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. For more information, see the Constellation Visualization section of the *Communications System Toolbox User’s Guide*.

Dialog Box



Signal constellation

A real or complex vector that lists the constellation points.

Output data type

The output data type can be set to `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

Setting this to `Fixed-point` or `User-defined` will enable fields in which you can further specify details. Setting this to `Inherit via back propagation`, sets the output data type and scaling to match the following block..

Output 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 signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer software. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set output fraction length 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 **Output 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.

Output 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 output fraction length to** parameter.

Supported Data Types

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

Port	Supported Data Types
	• $ufix(\lceil \log_2 M \rceil)$
Output	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

General QAM Demodulator Baseband

See Also

Rectangular QAM Modulator Baseband

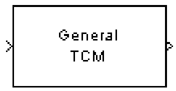
Purpose

Decode trellis-coded modulation data, mapped using arbitrary constellation

Library

TCM, in Digital Baseband sublibrary of Modulation

Description



The General TCM Decoder block uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using an arbitrary signal constellation.

The **Trellis structure** and **Signal constellation** parameters in this block should match those in the General TCM Encoder block, to ensure proper decoding. In particular, the **Signal constellation** parameter must be in set-partitioned order.

Input and Output Signals

This block accepts a column vector input signal containing complex numbers. The input signal must be `double` or `single`. The reset port signal must be `double` or `Boolean`. For information about the data types each block port supports, see “Supported Data Types” on page 2-356.

If the convolutional encoder described by the trellis structure represents a rate *k*/*n* code, then the General TCM Decoder block’s output is a binary column vector whose length is *k* times the vector length of the input signal.

Operation Modes

The block has three possible methods for transitioning between successive frames. The **Operation mode** parameter controls which method the block uses. This parameter also affects the range of possible values for the **Traceback depth** parameter, D.

- In `Continuous` mode, the block initializes all state metrics to zero at the beginning of the simulation, waits until it accumulates D symbols, and then uses a sequence of D symbols to compute each of the traceback paths. D can be any positive integer. At the end of each frame, the block saves its internal state metric for use with the next frame.

If you select **Enable the reset input port**, the block displays another input port, labeled `Rst`. This port receives an integer scalar signal. Whenever the value at the `Rst` port is nonzero, the block resets all state metrics to zero and sets the traceback memory to zero.

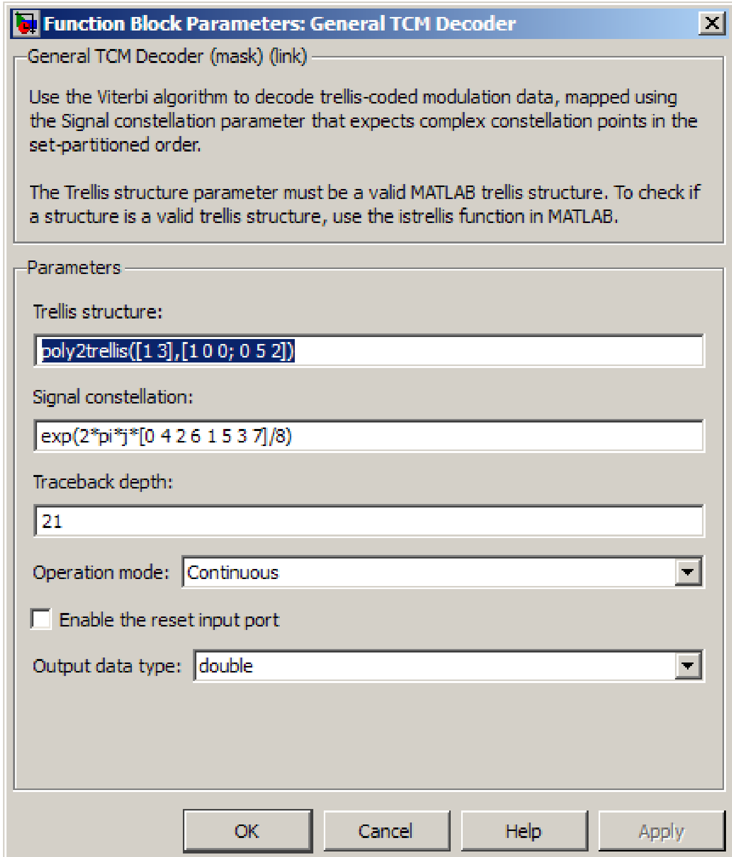
- In `Truncated` mode, the block treats each frame independently. The traceback path starts at the state with the lowest metric. D must be less than or equal to the vector length of the input.
- In `Terminated` mode, the block treats each frame independently. The traceback path always starts at the all-zeros state. D must be less than or equal to the vector length of the input. If you know that each frame of data typically ends at the all-zeros state, then this mode is an appropriate choice.

Decoding Delay

If you set **Operation mode** to `Continuous`, then this block introduces a decoding delay equal to **Traceback depth***k bits for a rate k/n convolutional code. The decoding delay is the number of zeros that precede the first decoded bit in the output.

The block incurs no delay for other values of **Operation mode**.

Dialog Box



Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Signal constellation

A complex vector that lists the points in the signal constellation in set-partitioned order.

Traceback depth
The number of trellis branches (equivalently, the number of symbols) the block uses in the Viterbi algorithm to construct each traceback path.

Operation mode
The operation mode of the Viterbi decoder. The choices are `Continuous`, `Truncated`, and `Terminated`.

Enable the reset input port
When you select this check box, the block has a second input port labeled `Rst`. Providing a nonzero value to this port causes the block to set its internal memory to the initial state before processing the input data. This field appears only if you set **Operation mode** to `Continuous`.

Output data type
Select the data type for the block output signal as `boolean` or `single`. By default, the block sets this to `double`.

Supported Data Types

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

Pair Block

General TCM Encoder

See Also

M-PSK TCM Decoder, Rectangular QAM TCM Decoder, `poly2trellis`

References

[1] Biglieri, E., D. Divsalar, P. J. McLane, and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001.

General TCM Encoder

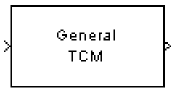
Purpose

Convolutionally encode binary data and map using arbitrary constellation

Library

TCM, in Digital Baseband sublibrary of Modulation

Description



The General TCM Encoder block implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal and mapping the result to an arbitrary signal constellation. The **Signal constellation** parameter lists the signal constellation points in set-partitioned order. This parameter is a complex vector with a length, M, equal to the number of possible output symbols from the convolutional encoder. (That is, $\log_2 M$ is equal to n for a rate k/n convolutional code.)

Input Signals and Output Signals

If the convolutional encoder represents a rate k/n code, then the General TCM Encoder block's input must be a binary column vector with a length of $L*k$ for some positive integer L.

This block accepts a binary-valued input signal. The output signal is a complex column vector of length L. For information about the data types each block port supports, see "Supported Data Types" on page 2-362.

Specifying the Encoder

To define the convolutional encoder, use the **Trellis structure** parameter. This parameter is a MATLAB structure whose format is described in the section "Trellis Description of a Convolutional Code" in the Communications System Toolbox documentation. You can use this parameter field in two ways:

- If you want to specify the encoder using its constraint length, generator polynomials, and possibly feedback connection polynomials, then use a `poly2trellis` command within the **Trellis structure** field. For example, to use an encoder with a constraint length of 7, code generator polynomials of 171 and 133 (in octal numbers), and a feedback connection of 171 (in octal), set the **Trellis structure** parameter to

```
poly2trellis(7,[171 133],171)
```

- If you have a variable in the MATLAB workspace that contains the trellis structure, then enter its name as the **Trellis structure** parameter. This way is faster because it causes Simulink software to spend less time updating the diagram at the beginning of each simulation, compared to the usage in the previous bulleted item.

The encoder registers begin in the all-zeros state. You can configure the encoder so that it resets its registers to the all-zeros state during the course of the simulation. To do this, set the `Operation mode` to **Reset on nonzero input via port**. The block then opens a second input port, labeled `Rst`. The signal at the `Rst` port is a scalar signal. When it is nonzero, the encoder resets before processing the data at the first input port.

Signal Constellations

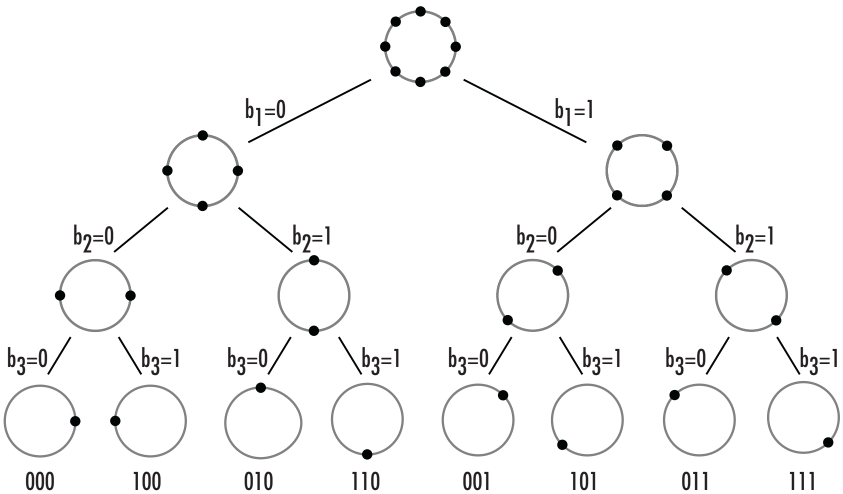
The trellis-coded modulation technique partitions the constellation into subsets called cosets so as to maximize the minimum distance between pairs of points in each coset.

Note When you set the **Signal constellation** parameter, you must ensure that the constellation vector is already in set-partitioned order. Otherwise, the block might produce unexpected or suboptimal results.

As an example, the diagram below shows one way to devise a set-partitioned order for the points for an 8-PSK signal constellation. The figure at the top of the tree is the entire 8-PSK signal constellation, while the eight figures at the bottom of the tree contain one constellation point each. Each level of the tree corresponds to a different bit in a binary sequence (b_3,b_2,b_1), while each branch in a given level of the tree corresponds to a particular value for that bit. Listing the constellation points using the sequence at the bottom of the tree leads to the vector

```
exp(2*pi*j*[0 4 2 6 1 5 3 7]/8)
```

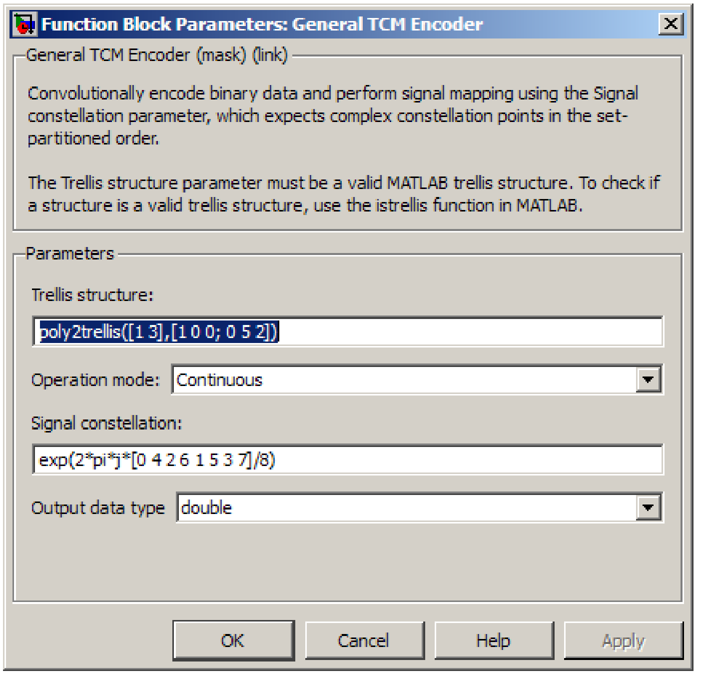
which is a valid value for the **Signal constellation** parameter in this block.



For other examples of signal constellations in set-partitioned order, see [1] or the reference pages for the M-PSK TCM Encoder and Rectangular QAM TCM Encoder blocks.

Coding Gains

Coding gains of 3 to 6 decibels, relative to the uncoded case can be achieved in the presence of AWGN with multiphase trellis codes [3].



Dialog Box

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Operation mode

In `Continuous` mode (default setting), the block retains the encoder states at the end of each frame, for use with the next frame.

In `Truncated (reset every frame)` mode, the block treats each frame independently. I.e., the encoder states are reset to all-zeros state at the start of each frame.

In `Terminate trellis by appending bits` mode, the block treats each frame independently. For each input frame, extra bits are used to set the encoder states to all-zeros state at the end of the frame. The output length is given by

$y = n \cdot (x + s) / k$, where x is the number of input bits, and

$s = \text{constraint length} - 1$ (or, in the case of multiple constraint lengths, s =`sum(ConstraintLength(i)-1)`). The block supports this mode for column vector input signals.

In `Reset on nonzero input via port` mode, the block has an additional input port, labeled `Rst`. When the `Rst` input is nonzero, the encoder resets to the all-zeros state.

Signal constellation
A complex vector that lists the points in the signal constellation in set-partitioned order.

Output data type
The output type of the block can be specified as a `single` or `double`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1)
Output	• Double-precision floating point • Single-precision floating point

Pair Block

General TCM Decoder

See Also

M-PSK TCM Encoder, Rectangular QAM TCM Encoder, `poly2trellis`

References

[1] Biglieri, E., D. Divsalar, P. J. McLane, and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001.

[3] Ungerboeck, G., "Channel Coding with Multilevel/Phase Signals", *IEEE Trans. on Information Theory*, Vol IT28, Jan. 1982, pp. 55–67.

GMSK Demodulator Baseband

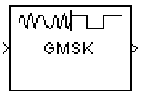
Purpose

Demodulate GMSK-modulated data

Library

CPM, in Digital Baseband sublibrary of Modulation

Description



The GMSK Demodulator Baseband block uses a Viterbi algorithm to demodulate a signal that was modulated using the Gaussian minimum shift keying method. The input to this block is a baseband representation of the modulated signal.

Integer-Valued Signals and Binary-Valued Signals

This block accepts a scalar-valued or column vector input signal with a data type of `single` or `double`. If you set the **Output type** parameter to `Integer`, then the block produces values of 1 and -1. If you set the **Output type** parameter to `Bit`, then the block produces values of 0 and 1.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input width must be an integer multiple of the **Samples per symbol** parameter value, and the input can be a column vector.

- When you set **Output type** to `Bit`, the output width is two times the number of input symbols.
- When you set **Output type** to `Integer`, the output width is the number of input symbols.

For a column vector input signal, the width of the input equals the product of the number of symbols and the value for the **Samples per symbol** parameter.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is the product of the input sample time and the **Samples per symbol** parameter value.

- When you set **Output type** to `Bit`, the output width equals the number of bits per symbol.
- When you set **Output type** to `Integer`, the output is a scalar.

Traceback Depth and Output Delays

Internally, this block creates a trellis description of the modulation scheme and uses the Viterbi algorithm. The **Traceback depth** parameter, D, in this block is the number of trellis branches used to construct each traceback path. D influences the output delay, which is the number of zero symbols that precede the first meaningful demodulated value in the output.

- When you set the **Rate options** parameter to `Allow multirate processing`, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking`, then the delay consists of D+1 zero symbols.
- When you set the **Rate options** parameter to `Enforce single-rate processing`, then the delay consists of D zero symbols.

The optimal **Traceback depth** parameter value is dependent on minimum squared Euclidean distance calculations. Alternatively, a typical value, dependent on the number of states, can be chosen using the “five-times-the-constraint-length” rule, which corresponds to $5 \cdot \log 2(numStates)$. The number of states is determined by the following equation:

$$numStates = \begin{Bmatrix} p \cdot 2^{(L-1)}, for\, even\, m \\ 2p \cdot 2^{(L-1)}, for\, odd\, m \end{Bmatrix}$$

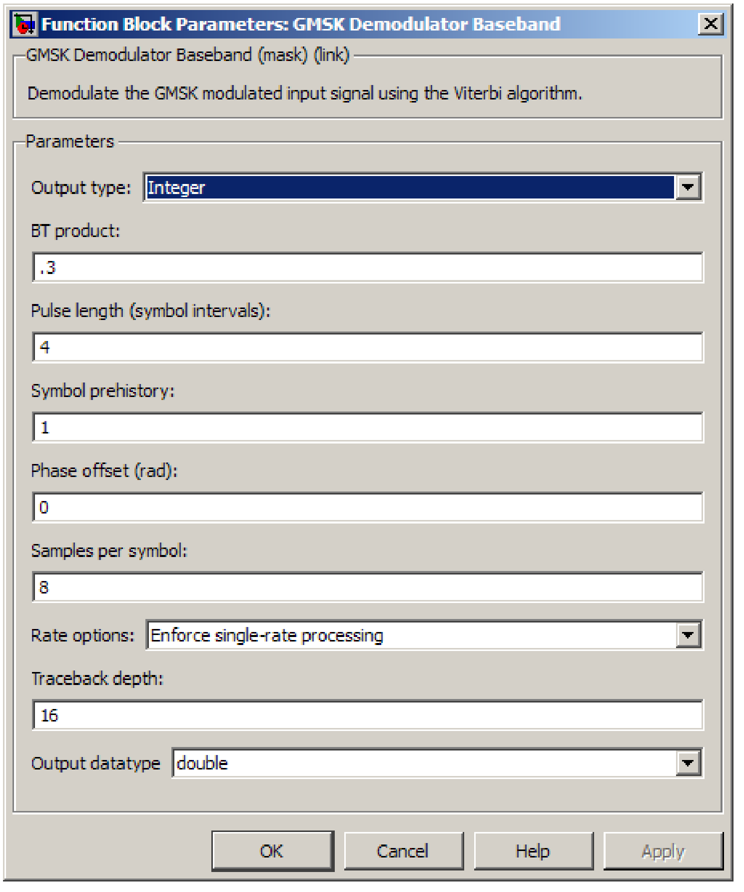
where:

- $h = m/p$ is the modulation index in proper rational form

- m = numerator of modulation index
- p = denominator of modulation index

- L is the Pulse length

Dialog Box



Output type

Determines whether the output consists of bipolar or binary values.

BT product

The product of bandwidth and time.

Pulse length (symbol intervals)

The length of the frequency pulse shape.

Symbol prehistory

The data symbols the modulator uses before the start of the simulation.

Phase offset (rad)

The initial phase of the modulated waveform.

Samples per symbol

The number of input samples that represent each modulated symbol, which must be a positive integer. For more information, see “Upsample Signals and Rate Changes” in *Communications System Toolbox User’s Guide*.

Rate options

Select the rate processing method for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width is the number of symbols (which is given by dividing the input length by the **Samples per symbol** parameter value when the **Output type** parameter is set to `Integer`).
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output period is the same as the symbol period and equals the product of the input period and the **Samples per symbol** parameter value.

> **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 *Communications System Toolbox Release Notes* for more information.

For more information, see Single-Rate Processing and Multirate Processing in the Description section of this page.

Traceback depth
The number of trellis branches that the GMSK Demodulator Baseband block uses to construct each traceback path.

Output data type
The output data type can be `boolean`, `int8`, `int16`, `int32`, or `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Boolean (When **Output type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (When **Output type** set to `Integer`)

Pair Block

GMSK Modulator Baseband

See Also

CPM Demodulator Baseband, Viterbi Decoder

References

[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg. *Digital Phase Modulation*. New York: Plenum Press, 1986.

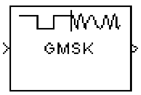
Purpose

Modulate using Gaussian minimum shift keying method

Library

CPM, in Digital Baseband sublibrary of Modulation

Description



The GMSK Modulator Baseband block modulates using the Gaussian minimum shift keying method. The output is a baseband representation of the modulated signal.

The **BT product** parameter represents bandwidth multiplied by time. This parameter is a nonnegative scalar. It is used to reduce the bandwidth at the expense of increased intersymbol interference. The **Pulse length** parameter measures the length of the Gaussian pulse shape, in symbol intervals. For an explanation of the pulse shape, see the work by Anderson, Aulin, and Sundberg among the references listed below. The frequency pulse shape is defined by the following equations.

$$g(t) = \frac{1}{2T}\left\{Q\left[2\pi B_b \frac{t-\frac{T}{2}}{\sqrt{\ln(2)}}\right] - Q\left[2\pi B_b \frac{t+\frac{T}{2}}{\sqrt{\ln(2)}}\right]\right\}$$

$$Q(t) = \int_t^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\tau^2/2} d\tau$$

For this block, an input symbol of 1 causes a phase shift of π/2 radians.

The group delay is the number of samples between the start of a filter's response and its peak. The group delay that the block introduces is **Pulse length**/2 * **Samples per symbol** (using a reference of output sample periods). For GMSK, **Pulse length** denotes the truncated frequency pulse length in symbols. The net delay effect at the receiver (demodulator) is due to the **Traceback depth** parameter, which in most cases would be larger than the group delay.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, then the block accepts values of 1 and -1.

When you set the **Input type** parameter to `Bit`, then the block accepts values of 0 and 1.

This block accepts a scalar-valued or column vector input signal. For a column vector input signal, the width of the output equals the product of the number of symbols and the value for the **Samples per symbol** parameter.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. In this mode, the input to the block can be multiple symbols.

- When you set **Input type** to `Integer`, the input can be a column vector, the length of which is the number of input symbols.
- When you set **Input type** to `Bit`, the input width must be an integer multiple of 2.

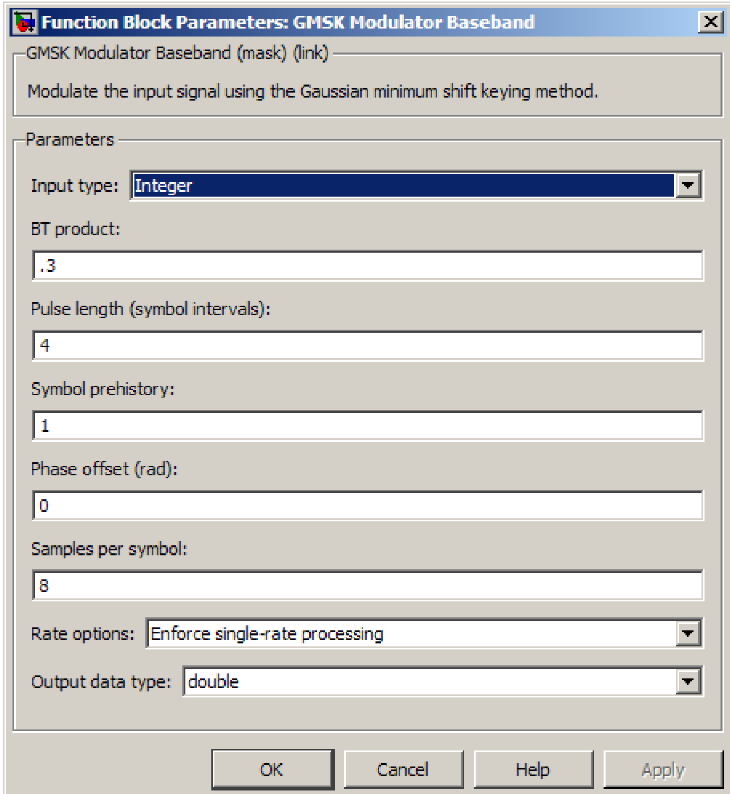
The output width equals the product of the number of input symbols and the **Samples per symbol** parameter value.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

- When you set **Input type** to `Integer`, the input must be a scalar.
- When you set **Input type** to `Bit`, the input width must equal the number of bits per symbol.

The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.



Dialog Box

Input type

Indicates whether the input consists of bipolar or binary values.

BT product

The product of bandwidth and time.

The block uses this parameter to reduce bandwidth at the expense of increased intersymbol interference. Enter a nonnegative scalar value for this parameter.

Pulse length (symbol intervals)

The length of the frequency pulse shape.

Symbol prehistory
A scalar or vector value that specifies the data symbols the block uses before the start of the simulation, in reverse chronological order. If it is a vector, then its length must be one less than the **Pulse length** parameter.

Phase offset (rad)
The initial phase of the output waveform, measured in radians.

Samples per symbol
The number of output samples that the block produces for each integer or bit in the input, which must be a positive integer. For all non-binary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see "Upsample Signals and Rate Changes" in *Communications System ToolboxUser's Guide*.

Rate options
Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

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 *Communications System Toolbox Release Notes* for more information.

Output data type

The output type of the block can be specified as a `single` or `double`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Boolean (When **Input type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (When **Input type** set to `Integer`)
Output	• Double-precision floating point • Single-precision floating point

Pair Block

GMSK Demodulator Baseband

See Also

CPM Modulator Baseband

References

[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg. *Digital Phase Modulation*. New York: Plenum Press, 1986.

Gold Sequence Generator

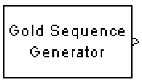
Purpose

Generate Gold sequence from set of sequences

Library

Sequence Generators sublibrary of Comm Sources

Description



The Gold Sequence Generator block generates a Gold sequence. Gold sequences form a large class of sequences that have good periodic cross-correlation properties.

This block can output sequences that vary in length during simulation. For more information about variable-size signals, see “Variable-Size Signal Basics” in the Simulink documentation.

Gold Sequences

The Gold sequences are defined using a specified pair of sequences u and v, of period $N = 2^n - 1$, called a *preferred pair*, as defined in “Preferred Pairs of Sequences” on page 2-377 below. The set $G(u, v)$ of Gold sequences is defined by

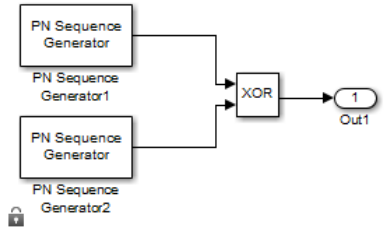
$$G(u,v) = \{u, v, u \oplus v, u \oplus Tv, u \oplus T^2 v, \ldots, u \oplus T^{N-1} v\}$$

where T represents the operator that shifts vectors cyclically to the left by one place, and $\oplus$ represents addition modulo 2. Note that G(u,v) contains $N + 2$ sequences of period N. The Gold Sequence Generator block outputs one of these sequences according to the block’s parameters.

Gold sequences have the property that the cross-correlation between any two, or between shifted versions of them, takes on one of three values: $-t(n)$, -1, or $t(n)$ - 2, where

$$t(n) = \begin{cases} 1 + 2^{(n+1)/2} & n \text{ even} \\ 1 + 2^{(n+2)/2} & n \text{ odd} \end{cases}$$

The Gold Sequence Generator block uses two PN Sequence Generator blocks to generate the preferred pair of sequences, and then XORs these sequences to produce the output sequence, as shown in the following diagram.



You can specify the preferred pair by the **Preferred polynomial [1]** and **Preferred polynomial [2]** parameters in the dialog for the Gold Sequence Generator block. These polynomials, both of which must have degree n, describe the shift registers that the PN Sequence Generator blocks use to generate their output. For more details on how these sequences are generated, see the reference page for the PN Sequence Generator block. You can specify the preferred polynomials using either of the following formats:

- A vector that lists the coefficients of the polynomial in descending order of powers. The first and last entries must be 1. Note that the length of this vector is one more than the degree of the generator polynomial.
- A vector containing the exponents of z for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`.

For example, the vectors `[5 2 0]` and `[1 0 0 1 0 1]` both represent the polynomial $z^5 + z^2 + 1$.

The following table provides a short list of preferred pairs.

n	N	Preferred Polynomial[1]	Preferred Polynomial[2]
5	31	`[5 2 0]`	`[5 4 3 2 0]`
6	63	`[6 1 0]`	`[6 5 2 1 0]`

n	N	Preferred Polynomial[1]	Preferred Polynomial[2]
7	127	[7 3 0]	[7 3 2 1 0]
9	511	[9 4 0]	[9 6 4 3 0]
10	1023	[10 3 0]	[10 8 3 2 0]
11	2047	[11 2 0]	[11 8 5 2 0]

The **Initial states[1]** and **Initial states[2]** parameters are vectors specifying the initial values of the registers corresponding to **Preferred polynomial [1]** and **Preferred polynomial [2]**, respectively. These parameters must satisfy these criteria:

- All elements of the **Initial states[1]** and **Initial states[2]** vectors must be binary numbers.
- The length of the **Initial states[1]** vector must equal the degree of the **Preferred polynomial[1]**, and the length of the **Initial states[2]** vector must equal the degree of the **Preferred polynomial[2]**.

Note At least one element of the **Initial states** vectors must be nonzero in order for the block to generate a nonzero sequence. That is, the initial state of at least one of the registers must be nonzero.

The **Sequence index** parameter specifies which sequence in the set $G(u, v)$ of Gold sequences the block outputs. The range of **Sequence index** is [-2, -1, 0, 1, 2, ..., 2^n-2]. The correspondence between **Sequence index** and the output sequence is given in the following table.

Sequence Index	Output Sequence
-2	u
-1	v

Sequence Index	Output Sequence
0	$u \oplus v$
1	$u \oplus Tv$
2	$u \oplus T^2 v$
...	...
2^n-2	$u \oplus T^{2^n - 2} v$

You can shift the starting point of the Gold sequence with the **Shift** parameter, which is an integer representing the length of the shift.

You can use an external signal to reset the values of the internal shift register to the initial state by selecting **Reset on nonzero input**. This creates an input port for the external signal in the Gold Sequence Generator block. The way the block resets the internal shift register depends on whether its output signal and the reset signal are sample-based or frame-based. The following example demonstrates the possible alternatives. See "Example: Resetting a Signal" on page 2-723 for an example.

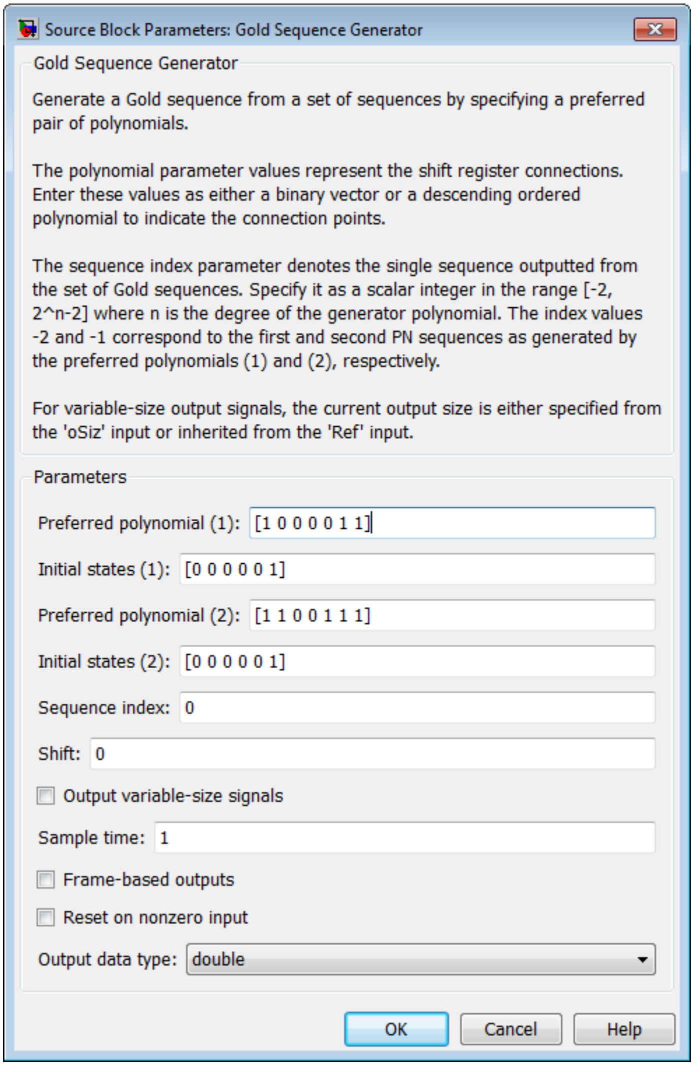
Preferred Pairs of Sequences

The requirements for a pair of sequences u, v of period $N = 2^n - 1$ to be a preferred pair are as follows:

- n is not divisible by 4
- v = u[q], where
  - q is odd
  - $q = 2^k+1$ or $q = 2^{2k}-2^k+1$
  - v is obtained by sampling every qth symbol of u

- $\gcd(n,k) = \begin{cases} 1 & n \equiv 1 \bmod 2 \\ 2 & n \equiv 2 \bmod 4 \end{cases}$

Dialog Box



Preferred polynomial[1]

Vector specifying the polynomial for the first sequence of the preferred pair.

Initial states[1]
Vector of initial states of the shift register for the first sequence of the preferred pair.

Preferred polynomial[2]
Vector specifying the polynomial for the second sequence of the preferred pair.

Initial states[2]
Vector of initial states of the shift register for the second sequence of the preferred pair.

Sequence index
Integer specifying the index of the output sequence from the set of sequences.

Shift
Integer scalar that determines the offset of the Gold sequence from the initial time.

Output variable-size signals
Select this check box if you want the output sequences to vary in length during simulation. The default selection outputs fixed-length signals.

Maximum output size source
Specify how the block defines maximum output size for a signal.

- When you select `Dialog parameter`, the value you enter in the **Maximum output size** parameter specifies the maximum size of the output. When you make this selection, the `oSiz` input port specifies the current size of the output signal and the block output inherits sample time from the input signal. The input value must be less than or equal to the **Maximum output size** parameter.
- When you select `Inherit from reference port`, the block output inherits sample time, maximum size, and current size from the variable-sized signal at the Ref input port.

This parameter only appears when you select **Output variable-size signals**. The default selection is `Dialog parameter`.

Maximum output size
Specify a two-element row vector denoting the maximum output size for the block. The second element of the vector must be `1` For example, [10 1] gives a 10-by-1 maximum sized output signal. This parameter only appears when you select **Output variable-size signals**.

Sample time
Period of each element of the output signal.

Frame-based outputs
Determines whether the output is frame-based or sample-based.

Samples per frame
The number of samples in a frame-based output signal. This field is active only if you select **Frame-based outputs**.

Reset on nonzero input
When selected, you can specify an input signal that resets the internal shift registers to the original values of the **Initial states** parameter

Output data type
The output type of the block can be specified as `boolean`, `double` or `Smallest unsigned integer`. By default, the block sets this to `double`.

When the parameter is set to `Smallest unsigned integer`, the output data type is selected based on the settings used in the Hardware Implementation pane of the Configuration Parameters dialog box of the model. If ASIC/FPGA is selected in the Hardware Implementation pane, the output data type is the ideal minimum one-bit size, i.e., ufix(1). For all other selections, it is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding to the size of a char (e.g., uint8).

See Also

Kasami Sequence Generator, PN Sequence Generator

References

[1] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

[2] Gold, R., "Maximal Recursive Sequences with 3-valued Recursive Cross-Correlation Functions," *IEEE Trans. Infor. Theory*, Jan., 1968, pp. 154-156.

[3] Gold, R., "Optimal Binary Sequences for Spread Spectrum Multiplexing, *IEEE Trans. Infor. Theory*, Oct., 1967, pp. 619-621.

[4] Sarwate, D.V., and M.B. Pursley, "Crosscorrelation Properties of Pseudorandom and Related Sequences," *Proc. IEEE*, Vol. 68, No. 5, May, 1980, pp. 583-619.

[5] Dixon, Robert, *Spread Spectrum Systems with Commercial Applications*, Third Edition, Wiley–Interscience, 1994.

Purpose

Generate Hadamard code from orthogonal set of codes

Library

Sequence Generators sublibrary of Comm Sources

Description

The Hadamard Code Generator block generates a Hadamard code from a Hadamard matrix, whose rows form an orthogonal set of codes. Orthogonal codes can be used for spreading in communication systems in which the receiver is perfectly synchronized with the transmitter. In these systems, the despreading operation is ideal, as the codes are decorrelated completely.

The Hadamard codes are the individual rows of a Hadamard matrix. Hadamard matrices are square matrices whose entries are +1 or -1, and whose rows and columns are mutually orthogonal. If N is a nonnegative power of 2, the N-by-N Hadamard matrix, denoted H_N, is defined recursively as follows.

$$H_1 = [1]$$

$$H_{2N} = \begin{bmatrix} H_N & H_N \\ H_N & -H_N \end{bmatrix}$$

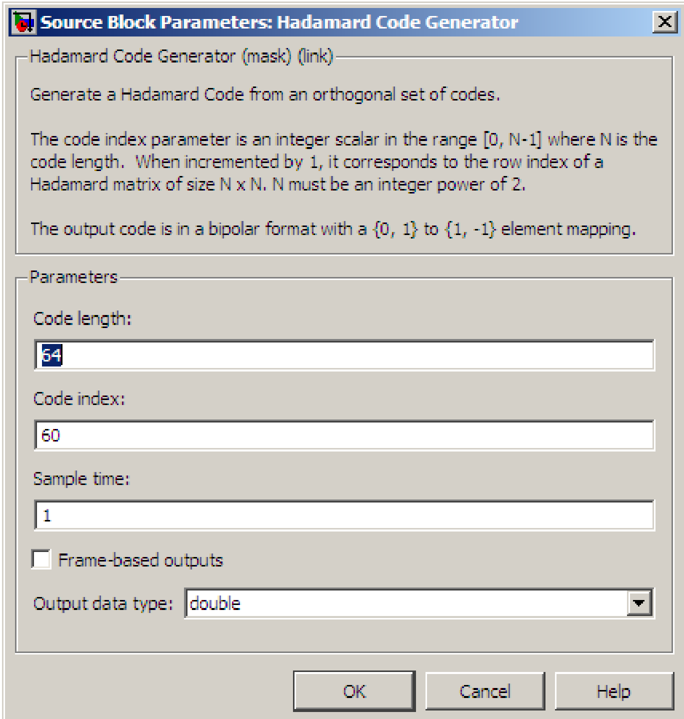
The N-by-N Hadamard matrix has the property that

$$H_N H_N^T = NI_N$$

where I_N is the N-by-N identity matrix.

The Hadamard Code Generator block outputs a row of H_N. The output is bipolar. You specify the length of the code, N, by the **Code length** parameter. The **Code length** must be a power of 2. You specify the index of the row of the Hadamard matrix, which is an integer in the range [0, 1, ... , N-1], by the **Code index** parameter.

Dialog Box



Code length
: A positive integer that is a power of two specifying the length of the Hadamard code.

Code index
: An integer between 0 and N-1, where N is the **Code length**, specifying a row of the Hadamard matrix.

Sample time
: A positive real scalar specifying the sample time of the output signal.

Frame-based outputs
: Determines whether the output is frame-based or sample-based.

Samples per frame
The number of samples in a frame-based output signal. This field is active only if you select **Frame-based outputs**.

Output data type
The output type of the block can be specified as an `int8` or `double`. By default, the block sets this to `double`.

See Also

OVSF Code Generator, Walsh Code Generator

Hamming Decoder

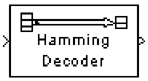
Purpose

Decode Hamming code to recover binary vector data

Library

Block sublibrary of Error Detection and Correction

Description



The Hamming Decoder block recovers a binary message vector from a binary Hamming codeword vector. For proper decoding, the parameter values in this block should match those in the corresponding Hamming Encoder block.

If the Hamming code has message length K and codeword length N, then N must have the form 2^M-1 for some integer M greater than or equal to 3. Also, K must equal N-M.

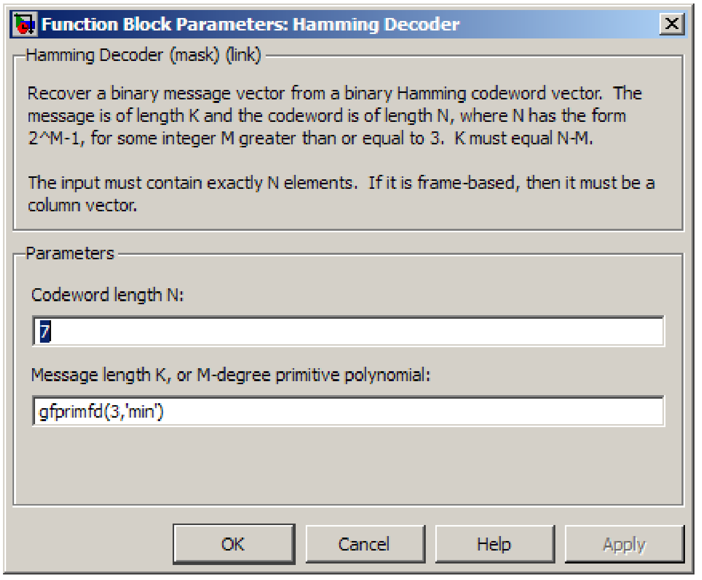
This block accepts a column vector input signal of length N. The output signal is a column vector of length K.

The coding scheme uses elements of the finite field GF(2^M). You can either specify the primitive polynomial that the algorithm should use, or you can rely on the default setting:

- To use the default primitive polynomial, simply enter N and K as the first and second dialog parameters, respectively. The algorithm uses `gfprimdf(M)` as the primitive polynomial for GF(2^M).
- To specify the primitive polynomial, enter N as the first parameter and a binary vector as the second parameter. The vector represents the primitive polynomial by listing its coefficients in order of ascending exponents. You can create primitive polynomials using the Communications System Toolbox `gfprimfd` function.

For information about the data types each block port supports, see the “Supported Data Type” on page 2-388 table on this page.

Dialog Box



Codeword length N

The codeword length N, which is also the input vector length.

Message length K, or M-degree primitive polynomial

Either the message length, which is also the output vector length; or a binary vector that represents a primitive polynomial for $GF(2^M)$.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point

Pair Block

Hamming Encoder

See Also

`hammgen` (Communications System Toolbox)

Hamming Encoder

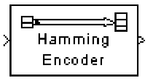
Purpose

Create Hamming code from binary vector data

Library

Block sublibrary of Error Detection and Correction

Description



The Hamming Encoder block creates a Hamming code with message length *K* and codeword length *N*. The number *N* must have the form 2^M-1, where *M* is an integer greater than or equal to 3. Then *K* equals *N-M*.

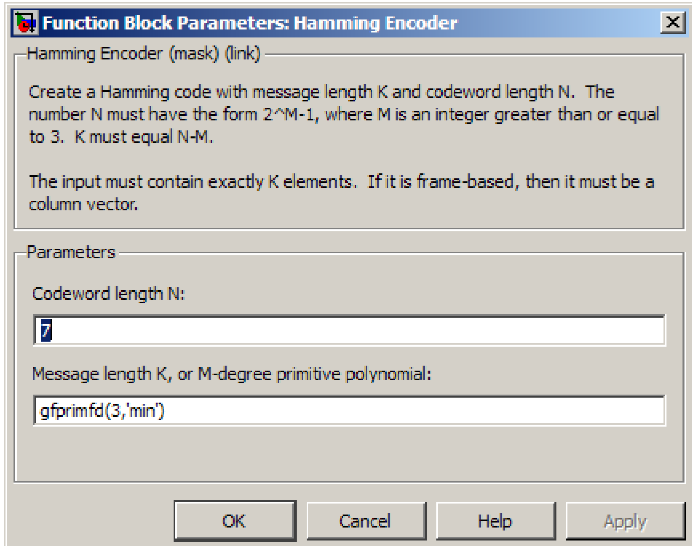
This block accepts a column vector input signal of length *K*. The output signal is a column vector of length *N*.

The coding scheme uses elements of the finite field GF(2^M). You can either specify the primitive polynomial that the algorithm should use, or you can rely on the default setting:

- To use the default primitive polynomial, simply enter *N* and *K* as the first and second dialog parameters, respectively. The algorithm uses `gfprimdf(M)` as the primitive polynomial for GF(2^M).
- To specify the primitive polynomial, enter *N* as the first parameter and a binary vector as the second parameter. The vector represents the primitive polynomial by listing its coefficients in order of ascending exponents. You can create primitive polynomials using the Communications System Toolbox `gfprimfd` function.

For information about the data types each block port supports, see the “Supported Data Type” on page 2-390 table on this page.

Dialog Box



Codeword length N
: The codeword length, which is also the output vector length.

Message length K, or M-degree primitive polynomial
: Either the message length, which is also the input vector length; or a binary vector that represents a primitive polynomial for GF(2^M).

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers

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

Pair Block

Hamming Decoder

See Also

`hammgen` (Communications System Toolbox)

Helical Deinterleaver

Purpose

Restore ordering of symbols permuted by helical interleaver

Library

Convolutional sublibrary of Interleaving

Description

The Helical Deinterleaver block permutes the symbols in the input signal by placing them in an array row by row and then selecting groups in a helical fashion to send to the output port.

The block uses the array internally for its computations. If C is the **Number of columns in helical array** parameter, then the array has C columns and unlimited rows. If N is the **Group size** parameter, then the block accepts an input of length $C \cdot N$ at each time step and inserts them into the next N rows of the array. The block also places the **Initial condition** parameter into certain positions in the top few rows of the array (not only to accommodate the helical pattern but also to preserve the vector indices of symbols that pass through the Helical Interleaver and Helical Deinterleaver blocks in turn).

The output consists of consecutive groups of N symbols. Counting from the beginning of the simulation, the block selects the kth output group in the array from column k mod C. The selection is helical because of the reduction modulo C and because the first symbol in the k^{th} group is in row 1+(k-1)*s, where s is the **Helical array step size** parameter.

This block accepts a column vector input signal containing $C \cdot N$ elements.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The data type of this output will be the same as that of the input signal.

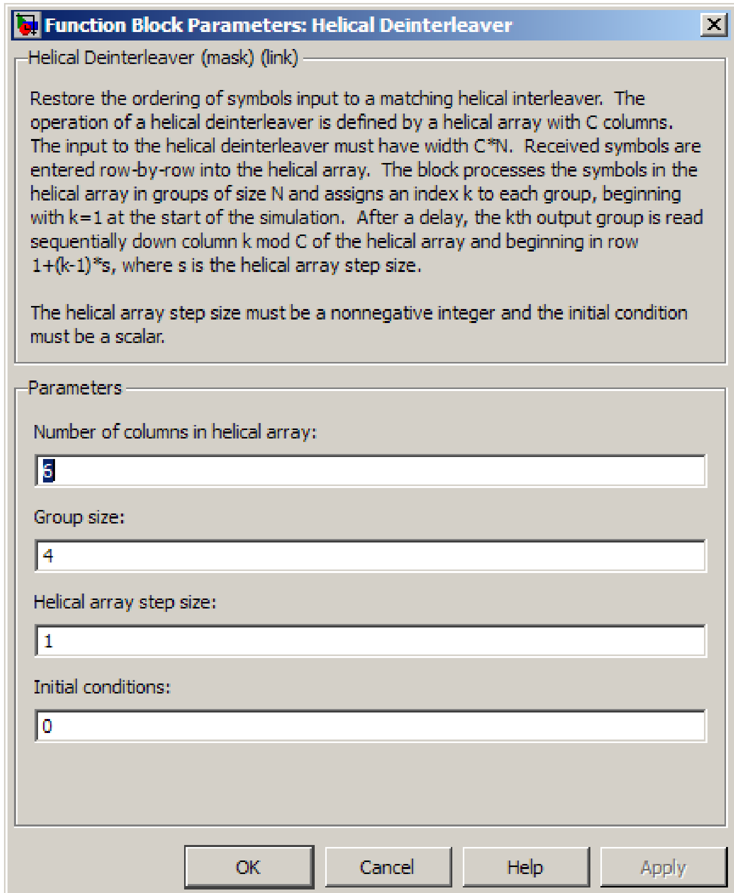
Delay of Interleaver-Deinterleaver Pair

After processing a message with the Helical Interleaver block and the Helical Deinterleaver block, the deinterleaved data lags the original message by

$$CN\left\lceil \frac{s(C-1)}{N} \right\rceil$$

samples. Before this delay elapses, the deinterleaver output is either the **Initial condition** parameter in the Helical Deinterleaver block or the **Initial condition** parameter in the Helical Interleaver block.

If your model incurs an additional delay between the interleaver output and the deinterleaver input, then the restored sequence lags the original sequence by the sum of the additional delay and the amount in the formula above. For proper synchronization, the delay between the interleaver and deinterleaver must be $\mathrm{m} \cdot C \cdot N$ for some nonnegative integer m. You can use the DSP System Toolbox Delay block to adjust delays manually, if necessary.



Dialog Box

Number of columns in helical array

The number of columns, C, in the helical array.

Group size

The size, N, of each group of symbols. The input width is C times N.

Helical array step size

The number of rows of separation between consecutive output groups as the block selects them from their respective columns of the helical array.

Initial conditions

A scalar that fills the array before the first input is placed.

Pair Block

Helical Interleaver

See Also

General Multiplexed Deinterleaver

References

[1] Berlekamp, E. R. and P. Tong. "Improved Interleavers for Algebraic Block Codes." U. S. Patent 4559625, Dec. 17, 1985.

Helical Interleaver

Purpose

Permute input symbols using helical array

Library

Convolutional sublibrary of Interleaving

Description

The Helical Interleaver block permutes the symbols in the input signal by placing them in an array in a helical fashion and then sending rows of the array to the output port.

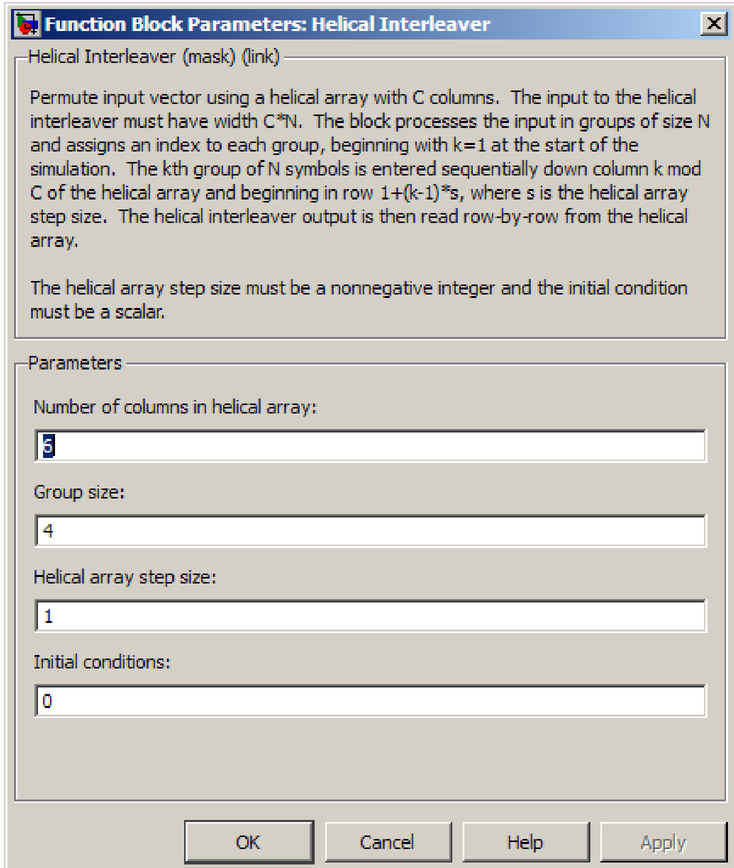
The block uses the array internally for its computations. If C is the **Number of columns in helical array** parameter, then the array has C columns and unlimited rows. If N is the **Group size** parameter, then the block accepts an input of length $C \cdot N$ at each time step and partitions the input into consecutive groups of N symbols. Counting from the beginning of the simulation, the block places the k^{th} group in the array along column k mod C. The placement is helical because of the reduction modulo C and because the first symbol in the k^{th} group is in row $1+(k-1) \cdot s$, where s is the **Helical array step size** parameter. Positions in the array that do not contain input symbols have default contents specified by the **Initial condition** parameter.

The block sends $C \cdot N$ symbols from the array to the output port by reading the next N rows sequentially. At a given time step, the output symbols might be the **Initial condition** parameter value, symbols from that time step's input vector, or symbols left in the array from a previous time step.

This block accepts a column vector input signal containing $C \cdot N$ elements.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The data type of this output will be the same as that of the input signal.

Dialog Box



Number of columns in helical array

The number of columns, C, in the helical array.

Group size

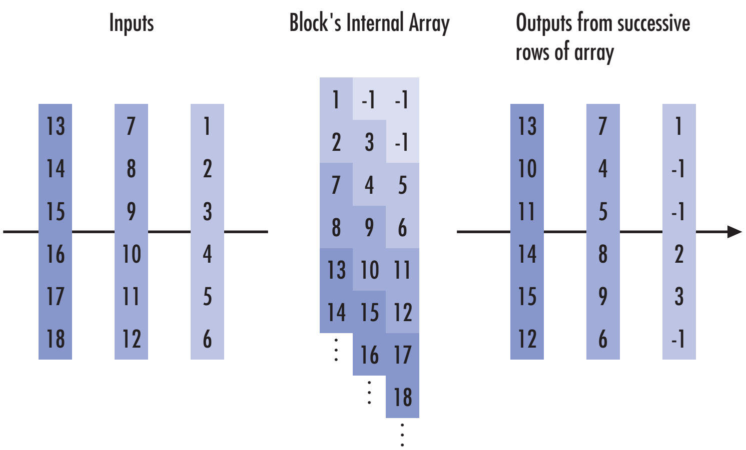
The size, N, of each group of input symbols. The input width is C times N.

Helical array step size
: The number of rows of separation between consecutive input groups in their respective columns of the helical array.

Initial conditions
: A scalar that fills the array before the first input is placed.

Examples

Suppose that C = 3, N = 2, the **Helical array step size** parameter is 1, and the **Initial condition** parameter is -1. After receiving inputs of `[1:6]'`, `[7:12]'`, and `[13:18]'`, the block's internal array looks like the schematic below. The coloring of the inputs and the array indicate how the input symbols are placed within the array. The outputs at the first three time steps are `[1; -1; -1; 2; 3; -1]`, `[7; 4; 5; 8; 9; 6]`, and `[13; 10; 11; 14; 15; 12]`. (The outputs are not color-coded in the schematic.)



Pair Block

Helical Deinterleaver

See Also

General Multiplexed Interleaver

References

[1] Berlekamp, E. R. and P. Tong. "Improved Interleavers for Algebraic Block Codes." U. S. Patent 4559625, Dec. 17, 1985.

Purpose

Shape input signal using ideal rectangular pulses

Library

Comm Filters

Description

The Ideal Rectangular Pulse Filter block upsamples and shapes the input signal using rectangular pulses. The block replicates each input sample N times, where N is the **Pulse length** parameter. After replicating input samples, the block can also normalize the output signal and/or apply a linear amplitude gain.

If the **Pulse delay** parameter is nonzero, then the block outputs that number of zeros at the beginning of the simulation, before starting to replicate any of the input values.

This block accepts a scalar, column vector, or matrix input signal. For information about the data types each block port supports, see the "Supported Data Type" on page 2-408 table on this page.

The vector size, the pulse length, and the pulse delay are mutually independent. They do not need to satisfy any conditions with respect to each other.

Single-Rate Processing

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 generate 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 ($M_o = M_i*L$), where L is the **Pulse length (number of samples)** parameter value.

Multirate Processing

When you set the **Rate options** parameter to `Allow multirate processing`, the input and output of the block are the same size. However, the sample rate of the output is L times faster than that of the input (i.e. the output sample time is 1/N times the input sample time). When the block is in multirate processing mode, you must also specify a value for the **Input processing** parameter:

- 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 processes 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.
- When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats an M_i-by-*N* matrix input as *N* independent channels. The block processes 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 ($T_{fo} = T_{fi}/L$).

Normalization Methods

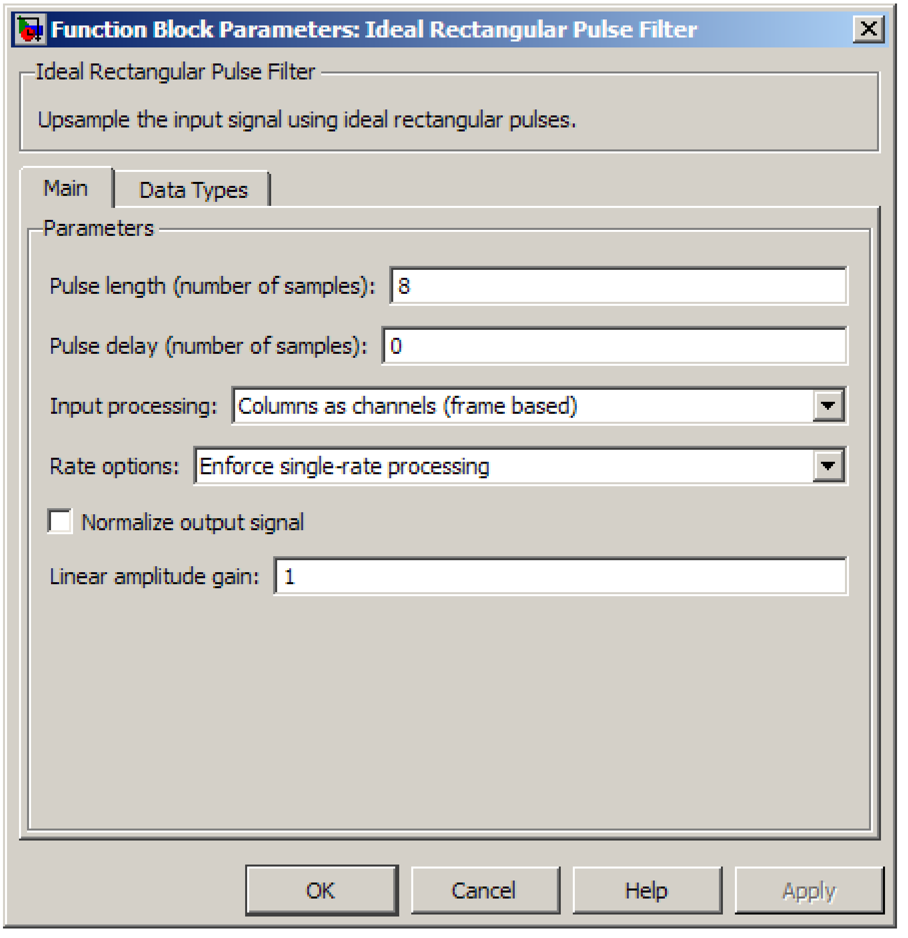
You determine the block's normalization behavior using the **Normalize output signal** and **Linear amplitude gain** parameters.

- If you clear **Normalize output signal**, then the block multiplies the set of replicated values by the **Linear amplitude gain** parameter. This parameter must be a scalar.
- If you select **Normalize output signal**, then the **Normalization method** parameter appears. The block scales the set of replicated values so that one of these conditions is true:
  - The sum of the samples in each pulse equals the original input value that the block replicated.
  - The energy in each pulse equals the energy of the original input value that the block replicated. That is, the sum of the squared samples in each pulse equals the square of the input value.

After the block applies the scaling specified in the **Normalization method** parameter, it multiplies the scaled signal by the constant scalar value specified in the **Linear amplitude gain** parameter.

The output is scaled by $\sqrt{N}$. If the output of this block feeds the input to the AWGN Channel block, specify the AWGN signal power parameter to be 1/N.

Dialog Box



Pulse length (number of samples)
The number of samples in each output pulse; that is, the number of times the block replicates each input value when creating the output signal.

Pulse delay (number of samples)
The number of zeros that appear in the output at the beginning of the simulation, before the block replicates any input values.

Input processing
Specify how the block processes the input signal. 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 *Communications System Toolbox Release Notes* for more information.

Rate options
Specify the method by which the block should upsample and shape the input signal. You can select one of the following options:

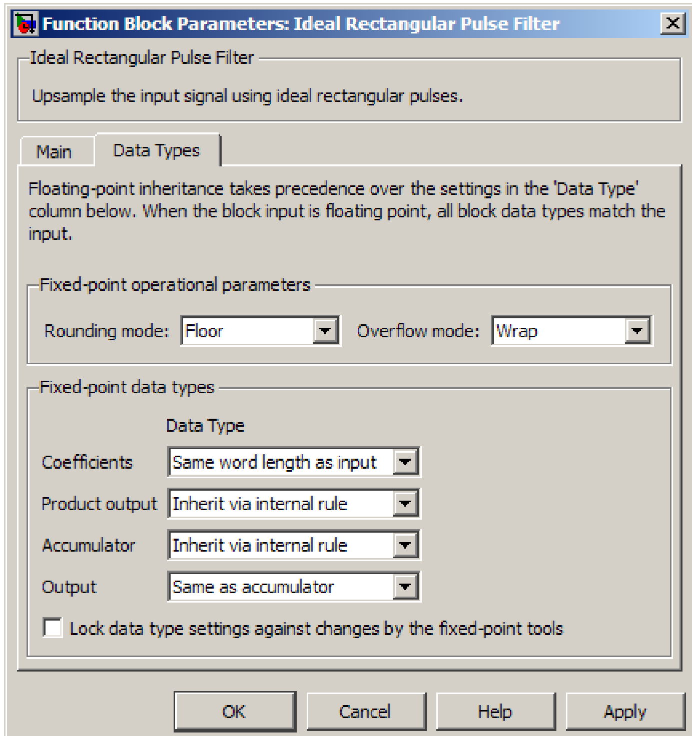
- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and processes the signal by increasing the output frame size by a factor of *L*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.

- `Allow multirate processing` — When you select this option, the block processes the signal such that the output sample rate is *L* times faster than the input sample rate.

Normalize output signal
: If you select this, then the block scales the set of replicated values before applying the linear amplitude gain.

Normalization method
: The quantity that the block considers when scaling the set of replicated values. Choices are `Sum of samples` and `Energy per pulse`. This field appears only if you select **Normalize method**.

Linear amplitude gain
: A positive scalar used to scale the output signal.



Rounding mode

Use this parameter to specify the rounding method to be used when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. The filter coefficients do not obey this parameter; they always round to `Nearest`.

For more information, see Rounding Modes in the DSP System Toolbox documentation or "Rounding Mode: Simplest" in the Fixed-Point Designer documentation.

Overflow mode

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

Coefficients

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). See "Filter Structure Diagrams" in *DSP System Toolbox Reference Guide* 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 are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits. If applicable, you are able to enter separate fraction lengths for the numerator and denominator coefficients.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. If applicable, you are able to enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.
- The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See "Filter Structure Diagrams" and "Multiplication Data Types" in *DSP System Toolbox Reference Guide* 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 are able to enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See "Filter Structure Diagrams" 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 are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

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

Lock scaling against changes by the autoscaling tool

Select this check box to prevent any fixed-point scaling you specify in the block mask from being overridden by the autoscaling tool in the Fixed-Point Tool.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Examples

If **Pulse length** is `4` and **Pulse delay** is the scalar `3`, then the table below shows how the block treats the beginning of a ramp (`1, 2, 3,...`) in several situations. (The values shown in the table do not reflect vector sizes but merely indicate numerical values.)

Normalization Method, If Any	Linear Amplitude Gain	First Several Output Values
None (**Normalize output signal** cleared)	1	0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3,...
None (**Normalize output signal** cleared)	10	0, 0, 0, 10, 10, 10, 10, 20, 20, 20, 20, 30, 30, 30, 30,...
`Sum of samples`	1	0, 0, 0, 0.25, 0.25, 0.25, 0.25, 0.5, 0.5, 0.5, 0.5, 0.75, 0.75, 0.75, 0.75,..., where 0.25*4=1
`Sum of samples`	10	0, 0, 0, 2.5, 2.5, 2.5, 2.5, 5, 5, 5, 5, 7.5, 7.5, 7.5, 7.5,...
`Energy per pulse`	1	0, 0, 0, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.5, 1.5, 1.5, 1.5,..., where (0.5)^2*4=1^2
`Energy per pulse`	10	0, 0, 0, 5, 5, 5, 5, 10, 10, 10, 10, 15, 15, 15, 15,...

See Also

Upsample, Integrate and Dump

Insert Zero

Purpose

Distribute input elements in output vector

Library

Sequence Operations

Description

The Insert Zero block constructs an output vector by inserting zeros among the elements of the input vector. The input signal can be real or complex. Both the input signal and the **Insert zero vector** parameter are column vector signals. The number of 1s in the **Insert zero vector** parameter must be evenly divisible by the input data length. If the input vector length is greater than the number of 1s in the **Insert zero vector** parameter, then the block repeats the insertion pattern until it has placed all input elements in the output vector.

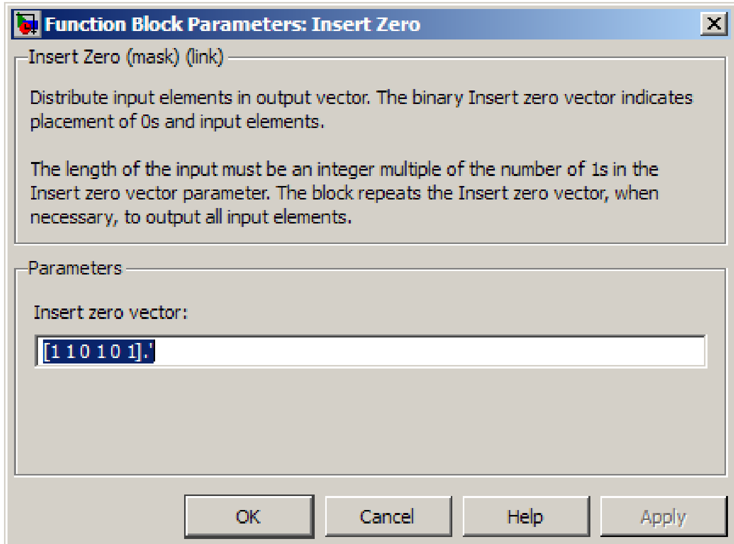
The block determines where to place the zeros by using the **Insert zero vector** parameter.

- For each 1 the block places the *next* element of the input vector in the output vector
- For each 0 the block places a 0 in the output vector

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

To implement punctured coding using the Puncture and Insert Zero blocks, use the same vector for the **Insert zero vector** parameter in this block and for the **Puncture vector** parameter in the Puncture block.

Dialog Box



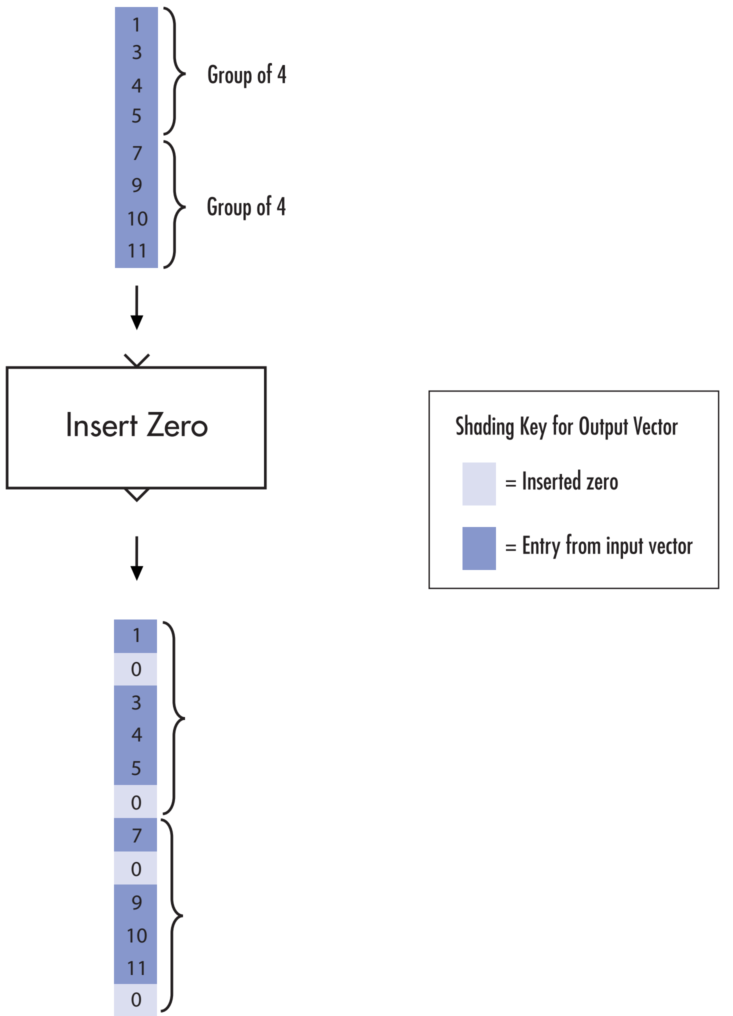
Insert zero vector

A binary vector with a pattern of 0s and 1s that indicate where the block places either 0s or input vector elements in the output vector.

Examples

If the **Insert zero vector** parameter is the six-element vector `[1;0;1;1;1;0]`, then the block inserts zeros after the first and last elements of each consecutive grouping of four input elements. It considers groups of four elements because the **Insert zero vector** parameter has four 1s.

The diagram below depicts the block's operation using this **Insert zero vector** parameter. Notice that the insertion pattern applies twice.


1
3
4
5
7
9
10
11
Group of 4
Group of 4
Insert Zero
1
0
3
4
5
0
7
0
9
10
11
0
Shading Key for Output Vector
= Inserted zero
= Entry from input vector

Compare this example with that on the reference page for the Puncture block.

See Also

Puncture

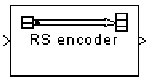
Purpose

Create Reed-Solomon code from integer vector data

Library

Block sublibrary of Error Detection and Correction

Description



The Integer-Input RS Encoder block creates a Reed-Solomon code with message length, K, and codeword length, (N - number of punctures). You specify both N and K directly in the block dialog. The symbols for the code are integers between 0 and 2^M-1, which represent elements of the finite field GF(2^M). Restrictions on M and N are described in "Restrictions on M and the Codeword Length N" on page 2-415 below. The difference N - K must be an even integer.

This block can output shortened codewords when N and K are appropriately specified. To specify output codewords that are shortened by a length S, N and K must be specified in the dialog box as $N_{full} - S$ and $K_{full} - S$, where N_{full} and K_{full} are the N and K of an unshortened code. If $S < (N_{full} + 1)/2$, the encoder can automatically determine the value of N_{full} and K_{full}. However, if $S \geq (N_{full} + 1)/2$, **Primitive polynomial** must be specified in order to properly define the extension field for the code.

The input and output are integer-valued signals that represent messages and codewords, respectively. This block accepts a column vector input signal with a length that is an integer multiple of K. The column vector output, with a length that is the same integer multiple of N, inherits its data type from the input signal. For information about the data types each block port supports, see the "Supported Data Type" on page 2-419 table on this page.

For more information on representing data for Reed-Solomon codes, see the section "Integer Format (Reed-Solomon Only)" in *Communications System Toolbox User's Guide*.

If the encoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

The default value of M is the smallest integer that is greater than or equal to log2(N+1), that is, `ceil(log2(N+1))`. You can change the value of M from the default by specifying the primitive polynomial for GF(2^M), as described in “Specifying the Primitive Polynomial” on page 2-415 below. If N is less than 2^M-1, the block uses a shortened Reed-Solomon code.

An (N, K) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (*not* bit errors) in each codeword.

Specifying the Primitive Polynomial

You can specify the primitive polynomial that defines the finite field GF(2^M), corresponding to the integers that form messages and codewords. To do so, first select **Specify primitive polynomial**. Then, in the **Primitive polynomial** field, enter a binary row vector that represents a primitive polynomial over GF(2) of degree M, in descending order of powers. For example, to specify the polynomial x^3+x+1, enter the vector `[1 0 1 1]`.

If you do not select **Specify primitive polynomial**, the block uses the default primitive polynomial of degree M = ceil(log2(N+1)). You can display the default polynomial by entering `primpoly(ceil(log2(N+1)))` at the MATLAB prompt.

Restrictions on M and the Codeword Length N

The restrictions on the degree M of the primitive polynomial and the codeword length N are as follows:

- If you do not select **Specify primitive polynomial**, N must lie in the range $3 < N \leq 2^{16}–1$.
- If you do select **Specify primitive polynomial**, N must lie in the range $3 \leq N \leq 2^M–1$ and M must lie in the range $3 \leq M \leq 16$.

Specifying the Generator Polynomial

You can specify the generator polynomial for the Reed-Solomon code. To do so, first select **Specify generator polynomial**. Then, in the **Generator polynomial** field, enter an integer row vector whose entries are between 0 and 2^M-1. The vector represents a polynomial, in descending order of powers, whose coefficients are elements of

$GF(2^M)$ represented in integer format. See the section "Integer Format (Reed-Solomon Only)" for more information about integer format. The generator polynomial must be equal to a polynomial with a factored form

$$g(x) = (x+A^b)(x+A^{b+1})(x+A^{b+2})...(x+A^{b+N-K-1})$$

where A is the primitive element of the Galois field over which the input message is defined, and b is an integer.

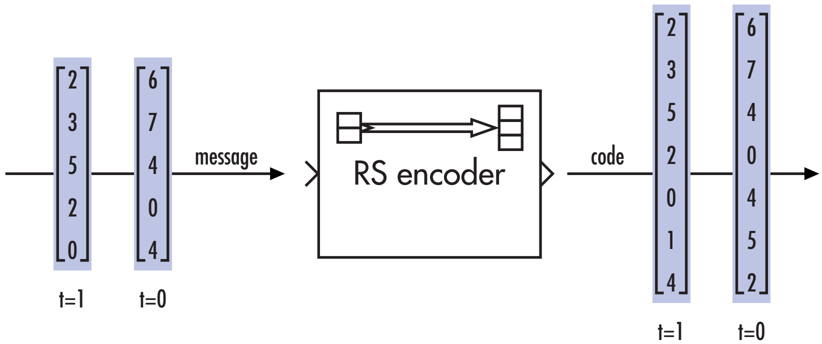
If you do not select **Specify generator polynomial**, the block uses the default generator polynomial, corresponding to b=1, for Reed-Solomon encoding. You can display the default generator polynomial by entering `rsgenpoly(N1,K1)`, where `N1 = 2^M-1` and `K1 = K+(N1-N)`, at the MATLAB prompt, if you are using the default primitive polynomial. If the **Specify primitive polynomial** box is selected, and you specify the primitive polynomial specified as `poly`, the default generator polynomial is `rsgenpoly(N1,K1,poly)`.

Puncture Codes

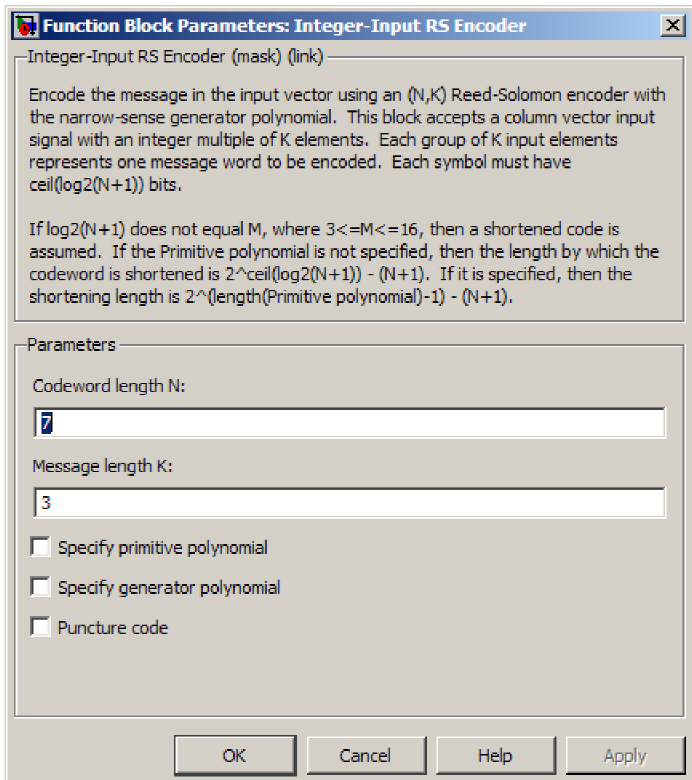
The block supports puncturing when you select the **Puncture code** parameter. This enables the **Puncture vector** parameter, which takes in a binary vector to specify the puncturing pattern. For a puncture vector, `1` represents that the data symbol passes unaltered, and `0` represents that the data symbol gets punctured, or removed, from the data stream. This convention is carried for both the encoder and the decoder. For more information, see "Shortening, Puncturing, and Erasures".

Examples

Suppose M = 3, N = 2^3-1 = 7, and K = 5. Then a message is a vector of length 5 whose entries are integers between 0 and 7. A corresponding codeword is a vector of length 7 whose entries are integers between 0 and 7. The following figure illustrates possible input and output signals to this block when **Codeword length N** is set to `7`, **Message length K** is set to `5`, and the default primitive and generator polynomials are used.


message
RS encoder
code
t=1
t=0
t=1
t=0

Dialog Box


Function Block Parameters: Integer-Input RS Encoder
Integer-Input RS Encoder (mask) (link)
Encode the message in the input vector using an (N,K) Reed-Solomon encoder with the narrow-sense generator polynomial. This block accepts a column vector input signal with an integer multiple of K elements. Each group of K input elements represents one message word to be encoded. Each symbol must have ceil(log2(N+1)) bits.
If log2(N+1) does not equal M, where 3<=M<=16, then a shortened code is assumed. If the Primitive polynomial is not specified, then the length by which the codeword is shortened is 2^ceil(log2(N+1)) - (N+1). If it is specified, then the shortening length is 2^(length(Primitive polynomial)-1) - (N+1).
Parameters
Codeword length N:
7
Message length K:
3
Specify primitive polynomial
Specify generator polynomial
Puncture code
OK
Cancel
Help
Apply

Codeword length N
The codeword length.

Message length K
The message length.

Specify primitive polynomial
Selecting this check box enables the **Primitive polynomial** parameter.

Primitive polynomial
Binary row vector representing the primitive polynomial in descending order of powers. When you provide a Primitive polynomial, the number of input bits must be an integer multiple of *K* times the order of the Primitive polynomial instead.

This parameter applies when only when you select **Specify primitive polynomial**.

Specify generator polynomial
Selecting this check box enables the **Generator polynomial** parameter.

Generator polynomial
This field is available only when **Specify generator polynomial** is selected.

Integer row vector, whose entries are in the range from 0 to 2^M-1, representing the generator polynomial in descending order of powers. Each coefficient is an element of the Galois field defined by the primitive polynomial.

Puncture code
Selecting this check box enables the **Puncture vector** parameter.

Puncture vector
This field is available only when **Puncture code** is selected.

A column vector of length *N-K*. In a puncture vector, `1` represents that the data symbol passes unaltered, and `0` represents that the data symbol gets punctured, or removed, from the data stream.

The default value is `[ones(2,1); zeros(2,1)]`.

This parameter applies only when you select **Puncture code**.

Supported Data Type

Port	Supported Data Types
In	• 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 • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

Integer-Output RS Decoder

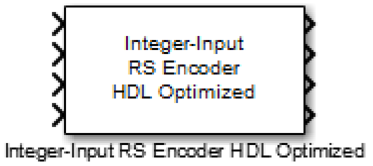
See Also

Binary-Input RS Encoder

Integer-Input RS Encoder HDL Optimized

Purpose Encode data using a Reed-Solomon encoder

Library Block sublibrary of Error Correction and Detection



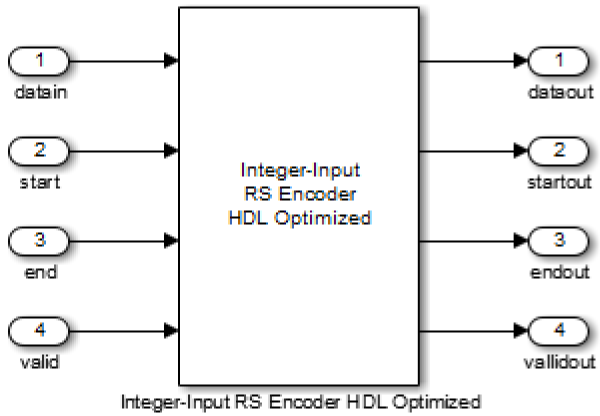
Description

Reed-Solomon encoding follows the same standards as any other cyclic redundancy code. The Integer-Input RS Encoder HDL Optimized block can be used to model many communication system Forward Error Correcting (FEC) codes.

For more about the Reed-Solomon encoder, see the Integer-Input RS Encoder block reference. For more information on representing data for Reed-Solomon codes, see "Integer Format (Reed-Solomon Only)".

Signal Attributes

The Integer-Input RS Encoder HDL Optimized block has four input ports and four output ports.



Port	Direction	Description	Data Type
`datain`	Input	Message data. Data width is less than or equal to the CRC length, and the CRC length should be divisible by the data width. For example, for CRC-CCITT/CRC-16, the valid data widths are 16, 8, 4, 2, and 1.	Column vector of double, Boolean, or ufix1
`start`	Input	Indicates the start of a frame of data.	Boolean or ufix1
`end`	Input	Indicates the end of a frame of data.	Boolean or ufix1
`valid`	Input	Indicates that input data is valid.	Boolean or ufix1

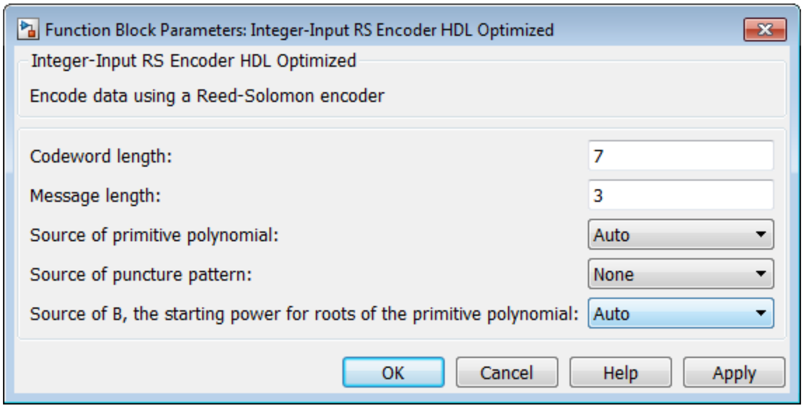
Port	Direction	Description	Data Type
dataout	Output	Message data with the checksum appended. The data width is the same as the input data port.	Column vector of double, Boolean, or ufix1
startout	Output	Indicates the start of a frame of data.	Boolean or ufix1
endout	Output	Indicates the end of a frame of data, including checksum.	Boolean or ufix1
validout	Output	Indicates that output data is valid.	Boolean or ufix1

Limitations

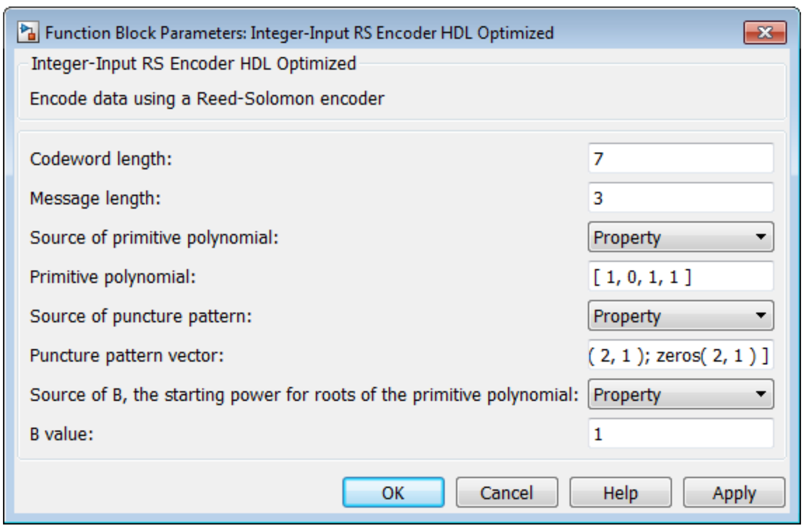
- The length of the code word `N` must be less than `2^16-1`. The number of parity symbols `N-K` must be a positive even integer. A shortened code is inferred anytime the number of input data samples is less than `2^M-1` for `M` between 3 and 16.
- The generator polynomial is not specified explicitly. However, it is defined by the code word length, the message length, and the B value for the starting exponent of the roots.
- For HDL code generation, the block does not handle double-precision input data. You can simulate using double-precision values, but if you attempt HDL code generation, you receive a error message.
- The Control Signals (start, end, valid) must be the Boolean datatype.

Block Dialog

Integer-Input RS Encoder HDL Optimized Block Mask, Default View



Integer-Input RS Encoder HDL Optimized Block Mask, Expanded View



Codeword length
: The codeword length.

Message length
: The message length.

Source of primitive polynomial
: Select `Property` to enable the **Primitive polynomial** parameter.

Primitive polynomial
: Binary row vector representing the primitive polynomial in descending order of powers. When you provide a primitive polynomial, the number of input bits must be an integer multiple of *K* times the order of the primitive polynomial instead.

  This parameter applies when only when `Property` is selected for **Primitive polynomial**.

Source of puncture pattern
: Select `Property` to enable the **Puncture pattern vector** parameter.

Puncture pattern vector
: A column vector of length *N*-*K*. In a puncture vector, `1` represents that the data symbol passes unaltered. The value `0` represents that the data symbol is punctured, or removed from the data stream.

  The default value is `[ones(2,1); zeros(2,1)]`.

  This field is available only when `Property` is selected for **Source of puncture pattern**.

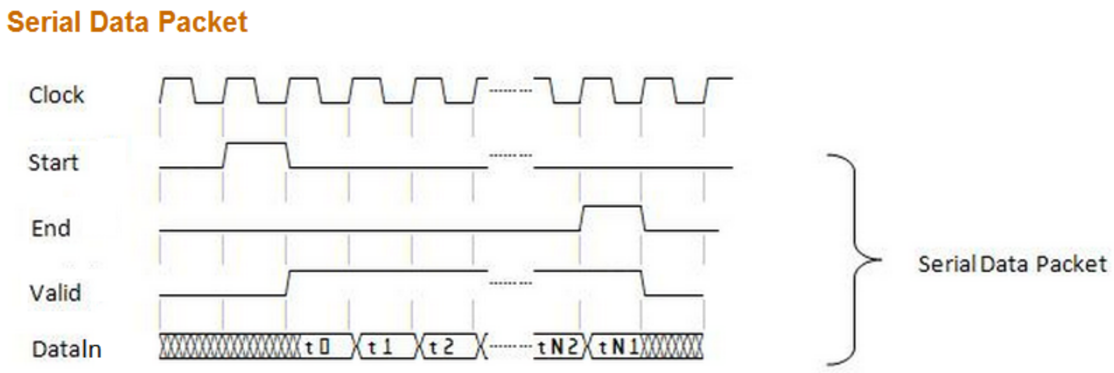
Source of B, the starting power for roots of the primitive polynomial
: Select `Property` to enable the **B value** parameter.

B value
: The starting exponent of the roots.

This field is available only when you select `Property` for **Source of B, the starting power for roots of the primitive polynomial**.

Algorithm

Timing Diagram



Pair Block

Integer-Output RS Decoder HDL Optimized

See Also

Integer-Input RS Encoder

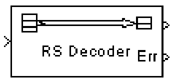
Purpose

Decode Reed-Solomon code to recover integer vector data

Library

Block sublibrary of Error Detection and Correction

Description



The Integer-Output RS Decoder block recovers a message vector from a Reed-Solomon codeword vector. For proper decoding, the parameter values in this block should match those in the corresponding Integer-Input RS Encoder block.

The Reed-Solomon code has message length, K, and codeword length, (N - number of punctures). You specify both N and K directly in the block dialog. The symbols for the code are integers between 0 and 2^M-1, which represent elements of the finite field GF(2^M). Restrictions on M and N are described in "Restrictions on M and the Codeword Length N" on page 2-415 below. The difference N - K must be an even integer.

This block can output shortened codewords when N and K are appropriately specified. To specify output codewords that are shortened by a length S, N and K must be specified in the dialog box as $N_{full} - S$ and $K_{full} - S$, where N_{full} and K_{full} are the N and K of an unshortened code. If $S < (N_{full} + 1)/2$, the encoder can automatically determine the value of N_{full} and K_{full}. However, if $S \geq (N_{full} + 1)/2$, **Primitive polynomial** must be specified in order to properly define the extension field for the code.

The input and output are integer-valued signals that represent codewords and messages, respectively. This block accepts a column vector input signal with a length that is an integer multiple of (N - number of punctures). The output signal is a column vector with a length that is the same integer multiple of K. The block inherits the output data type from the input data type. For information about the data types each block port supports, see the "Supported Data Type" on page 2-430 table on this page.

For more information on representing data for Reed-Solomon codes, see the section "Integer Format (Reed-Solomon Only)" in the *Communications System Toolbox User's Guide*.

If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

The default value of M is `ceil(log2(N+1))`, that is, the smallest integer greater than or equal to log2(N+1). You can change the value of M from the default by specifying the primitive polynomial for GF(2^M), as described in "Specifying the Primitive Polynomial" on page 2-415 below. If N is less than 2^M-1, the block uses a shortened Reed-Solomon code.

You can also specify the generator polynomial for the Reed-Solomon code, as described in "Specifying the Generator Polynomial" on page 2-415.

An (N, K) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (*not* bit errors) in each codeword.

The second output is the number of errors detected during decoding of the codeword. A -1 indicates that the block detected more errors than it could correct using the coding scheme. An (N,K) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (*not* bit errors) in each codeword. The data type of this output is also inherited from the input signal.

You can disable the second output by deselecting **Output number of corrected errors.** This removes the block's second output port.

In the case of a decoder failure, the message portion of the decoder input is returned unchanged as the decoder output.

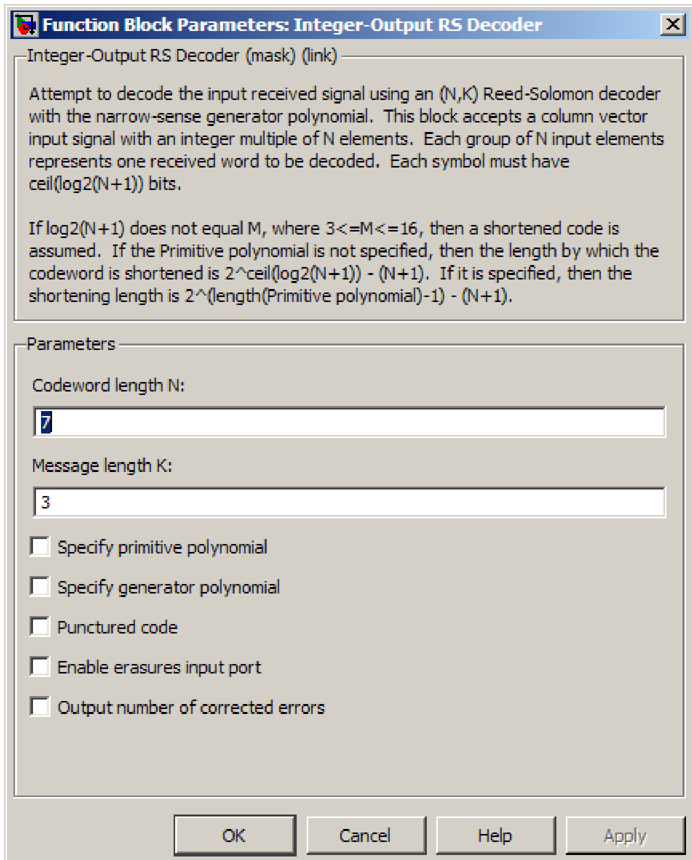
The sample times of the input and output signals are equal.

Punctured Codes

This block supports puncturing when you select the **Punctured code** parameter. This selection enables the **Puncture vector** parameter, which takes in a binary vector to specify the puncturing pattern. For a puncture vector, `1` represents that the data symbol passes unaltered, and `0` represents that the data symbol gets punctured, or removed, from the data stream. This convention is carried for both the encoder and the decoder. For more information, see "Shortening, Puncturing, and Erasures".

Note 1s and 0s have precisely opposite meanings for the puncture and erasure vectors. For an erasure vector, 1 means that the data symbol is to be replaced with an erasure symbol, and 0 means that the data symbol is passed unaltered. This convention is carried for both the encoder and the decoder.

Dialog Box



Codeword length N
The codeword length.

Message length K
The message length.

Specify primitive polynomial
Selecting this check box enables the field **Primitive polynomial**.

Primitive polynomial
This parameter applies only when you select **Specify primitive polynomial**.

Binary row vector representing the primitive polynomial in descending order of powers.

Specify generator polynomial
Selecting this check box enables the field **Generator polynomial**.

Generator polynomial
Integer row vector, whose entries are in the range from 0 to 2^M-1, representing the generator polynomial in descending order of powers. Each coefficient is an element of the Galois field defined by the primitive polynomial.

This parameter applies only when you select **Specify generator polynomial**.

Puncture code
Selecting this check box enables the field **Puncture vector**.

Puncture vector
A column vector of length *N-K*. In the **Puncture vector**, a value of `1` represents that the data symbol passes unaltered, and `0` represents that the data symbol gets punctured, or removed, from the data stream.

The default value is `[ones(2,1); zeros(2,1)]`.

This parameter applies only when you select **Puncture code**.

Enable erasures input port

Selecting this check box will open the port, `Era`. This port accepts a binary column vector input signal with the same size as the codeword.

Erasure values of `1` represents symbols in the same position in the codeword that get erased, and values of `0` represent symbols that do not get erased.

Output number of corrected errors

When you select this check box, the block outputs the number of corrected errors in each word through a second output port. A decoding failure occurs when a certain word in the input contains more than (N-K)/2 errors. A value of `-1` indicates a decoding failure in the corresponding position in the second output vector.

Algorithm

This block uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the references listed below.

Supported Data Type

Port	Supported Data Types
In	• 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 • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Era	• Double-precision floating point • Boolean
Err	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • If the input is uint8, uint16, or uint32, then the number of errors output datatype is int8, int16, or int32, respectively.

Pair Block

Integer-Input RS Encoder

References

[1] Wicker, Stephen B., *Error Control Systems for Digital Communication and Storage*, Upper Saddle River, N.J., Prentice Hall, 1995.

[2] Berlekamp, Elwyn R., *Algebraic Coding Theory*, New York, McGraw-Hill, 1968.

[3] Clark, George C., Jr., and J. Bibb Cain, *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

See Also

Binary-Output RS Decoder

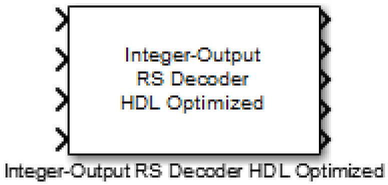
Integer-Output RS Decoder HDL Optimized

Purpose

Decode data using a Reed-Solomon decoder

Library

Block sublibrary of Error Correction and Detection



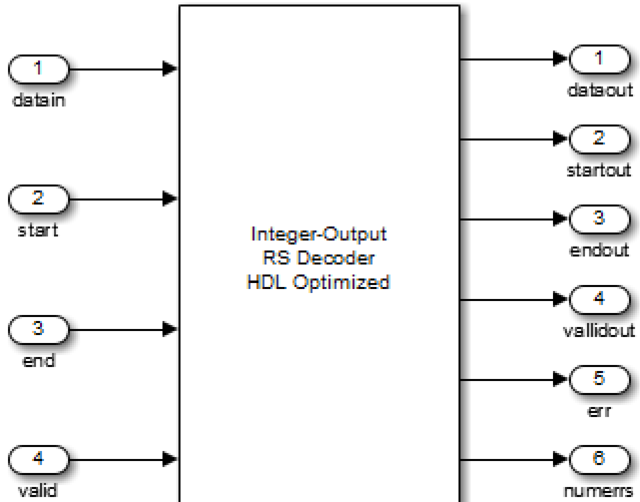
Description

Reed-Solomon encoding follows the same standards as any other cyclic redundancy code. The Integer-Output RS Decoder HDL Optimized block can be used to model many communication system Forward Error Correcting (FEC) codes.

For more about the Reed-Solomon decoder, see the Integer-Output RS Decoder block reference. For more information on representing data for Reed-Solomon codes, see "Integer Format (Reed-Solomon Only)".

Signal Attributes

The Integer-Output RS Decoder HDL Optimized block has four input ports and six output ports (5 required, 1 optional).



Integer-Output RS Decoder HDL Optimized

Port	Direction	Description	Data Type
datain	Input	Message data. Data width is less than or equal to the CRC length, and the CRC length should be divisible by the data width. For example, for CRC-CCITT/CRC-16, the valid data widths are 16, 8, 4, 2, and 1.	Must be an integer (uint8, uint16, uint32) or fi(). Doubles are allowed for simulation but not for HDL code generation.
start	Input	Indicates the start of a frame of data.	Boolean

Port	Direction	Description	Data Type
end	Input	Indicates the end of a frame of data.	Boolean
valid	Input	Indicates that input data is valid.	Boolean
dataout	Output	Message data with the checksum appended. The data width is the same as the input data port.	Must be an integer (uint8, uint16, uint32) or fi(). Doubles are allowed for
startout	Output	Indicates the start of a frame of data.	Boolean
endout	Output	Indicates the end of a frame of data, including checksum.	Boolean
validout	Output	Indicates that output data is valid.	Boolean
err	Output	Indicates the corruption of the received data when error is high.	Boolean in and out
numerrs	Output	Optional.	Always a uint8 output

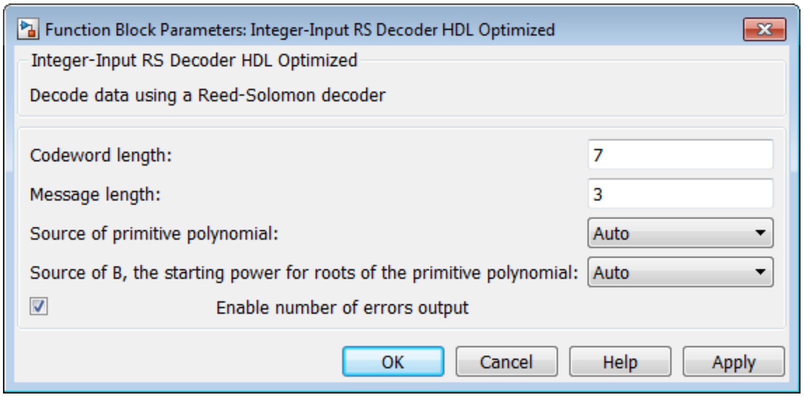
Limitations

- The length of the code word N must be less than 2^16-1. The number of parity symbols N-K must be a positive even integer. A shortened code is inferred when the number of valid data samples between start and is less than the codeword length.

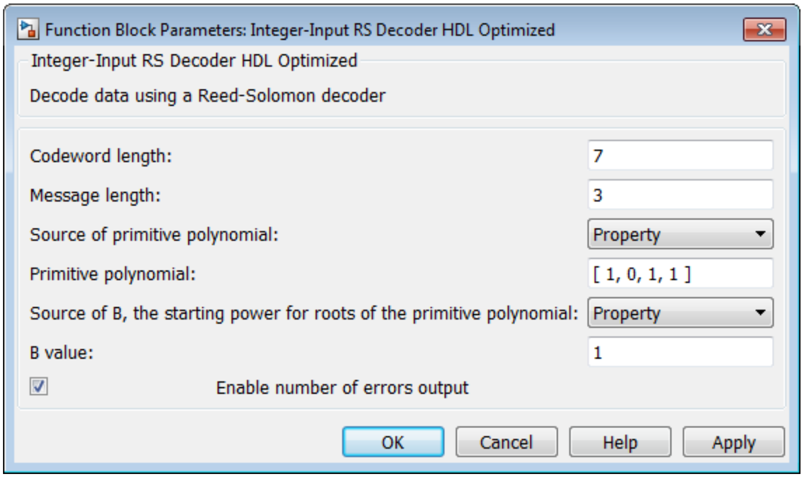
- The generator polynomial is not specified explicitly. However, it is defined by the code word length, the message length, and the B value for the starting exponent of the roots.
- For HDL code generation, the block does not handle double-precision floating point datatype numbers. You can simulate using double-precision values, but if you attempt HDL code generation, you receive a error message.

Dialog Box

Integer-Ouput RS Decoder HDL Optimized Block Mask, Default View



Integer-Output RS Decoder HDL Optimized Block Mask, Expanded View



Codeword length
: The codeword length.

Message length
: The message length.

Source of primitive polynomial
: Select `Property` to enable the **Primitive polynomial** parameter.

Primitive polynomial
: Binary row vector representing the primitive polynomial in descending order of powers. When you provide a primitive polynomial, the number of input bits must be an integer multiple of K times the order of the primitive polynomial instead.

  This parameter applies when only when `Property` is selected for **Primitive polynomial**.

Source of B, the starting power for roots of the primitive polynomial
: Select `Property` to enable the **B value** parameter.

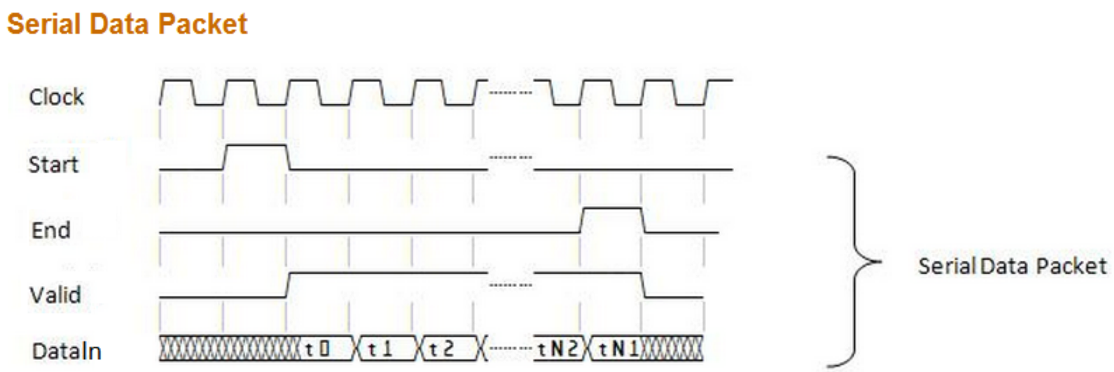
B value
The starting exponent of the roots.

This field is available only when you select `Property` for **Source of B, the starting power for roots of the primitive polynomial**.

Enable number of errors output
Check this box to enable the number of errors output port.

Algorithm

Timing Diagram



Pair Block

Integer-Input RS Encoder HDL Optimized

See Also

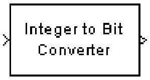
Integer-Output RS Decoder

Integer to Bit Converter

Purpose Map vector of integers to vector of bits

Library Utility Blocks

Description



The Integer to Bit Converter block maps each integer (or fixed-point value) in the input vector to a group of bits in the output vector.

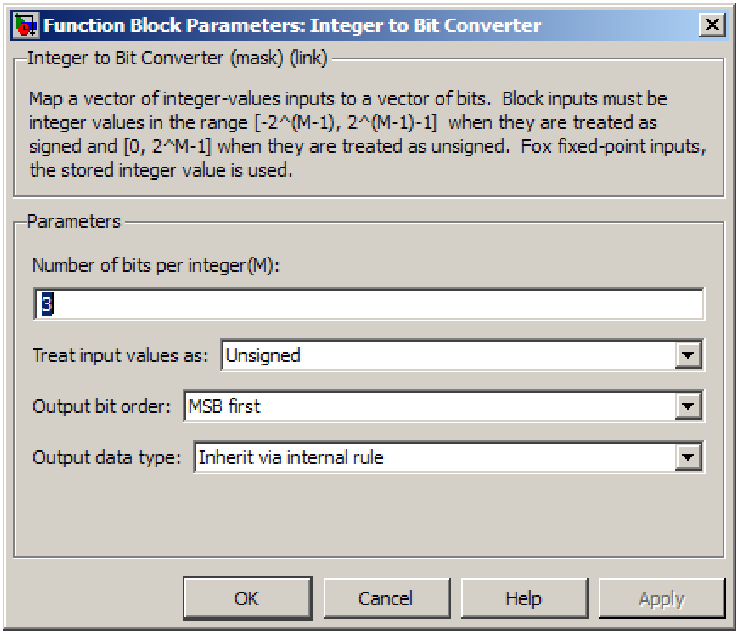
The block maps each integer value (or stored integer when you use a fixed point input) to a group of *M* bits, using the selection for the **Output bit order** to determine the most significant bit. The resulting output vector length is *M* times the input vector length.

When you set the **Number of bits per integer** parameter to *M* and **Treat input values as** to `Unsigned`, then the input values must be between 0 and 2^{M}-1. When you set **Number of bits per integer** to *M* and **Treat input values as** to `Signed`, then the input values must be between -2^{M-1} and 2^{M-1}-1. During simulation, the block performs a run-time check and issues an error if any input value is outside of the appropriate range. When the block generates code, it does not perform this run-time check.

This block is single-rate and single-channel. It accepts a length *N* column vector or a scalar-valued ($N = 1$) input signal and outputs a length $N \cdot M$ column vector.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, `double`, and `fixed point`.

Dialog Box



Number of bits per integer
: The number of bits the block uses to represent each integer of the input. This parameter must be an integer between 1 and 32.

Treat input values as
: Indicate if the integer value input ranges should be treated as signed or unsigned. The default setting is `Unsigned`.

Output bit order
: Define whether the first bit of the output signal is the most significant bit (MSB) or the least significant bit (LSB). The default selection is `MSB first`.

Output data type
: You can choose the following **Output data type** options:

- `Inherit via internal rule`
- `Smallest integer`

- `Same as input`
- `double`
- `single`
- `uint8`
- `uint16`
- `uint32`

The default selection for this parameter is `Inherit via internal rule`.

When the parameter is set to `Inherit via internal rule`, the block determines the output data type based on the input data type.

- If the input signal is floating-point (either `single` or `double`), the output data type is the same as the input data type.
- If the input data type is not floating-point, the output data type is determined as if the parameter is set to `Smallest integer`.

When the parameter is set to `Smallest integer`, the block selects the output data type based on settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- If you select ASIC/FPGA, the output data type is the ideal one-bit size; ufix1.
- For all other selections, the output data type is an unsigned integer with the smallest available word length, as defined in the Hardware Implementation settings (e.g. uint8)

Examples

Fixed-Point Integer To Bit and Bit To Integer Conversion (Audio Scrambling and Descrambling Example)

Overview

This example illustrates how to use the Bit to Integer and Integer to Bit Converter blocks with fixed-point signals.

This example uses a simplified audio scrambler configuration and a 16-bit, fixed-point digital audio source, which is recorded speech. The left-side of the model represents the audio scrambler subsystem and the right-side represents the descrambler subsystem.

Opening the Model

You can open the model by typing `doc_audioscrambler` at the MATLAB command line.

Structure

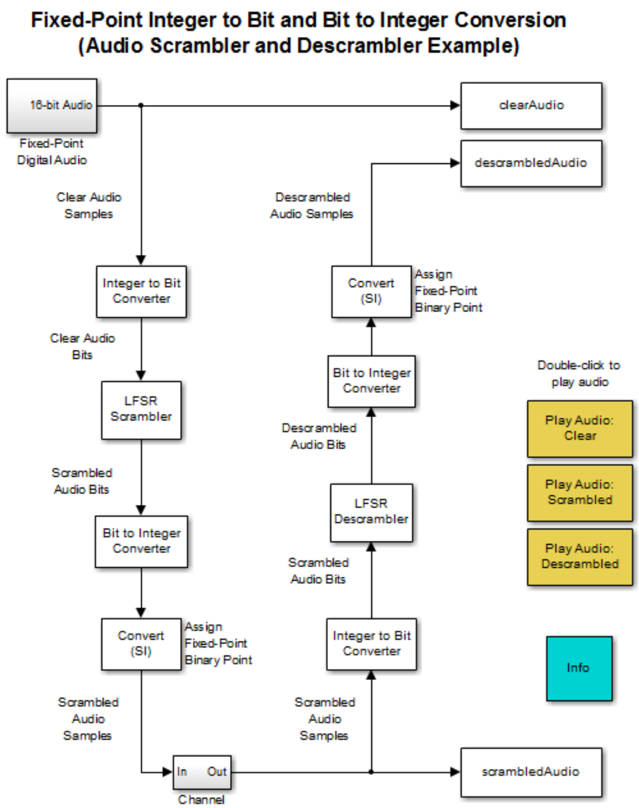
In the audio scrambler subsystem, the Integer to Bit Converter block unpacks each 16-bit audio sample into a binary, 1-bit signal. The binary signal passes to a linear feedback shift register (LFSR) scrambler, which randomizes the bits in a controllable way, thereby scrambling the signal. The Communications System Toolbox Scrambler block is used in the LFSR implementation. From the LFSR, the scrambled audio bits pass to the Bit to Integer Converter block. This block packs the scrambled 1-bit samples into 16-bit audio samples. The audio samples pass to the Data Type Conversion block, which converts the integer-based audio samples back into fixed-point samples.

The fixed-point samples pass from the scrambler subsystem to a channel. The channel sends the samples to the descrambler subsystem. For illustrative purposes, this example uses a noiseless channel. In an actual system, a channel may introduce noise. Removing such noise requires a more sophisticated design.

In the audio descrambler subsystem, the Integer to Bit Converter block unpacks each 16-bit audio sample into a binary, 1-bit signal. The binary signal passes to a linear feedback shift register (LFSR) descrambler, which randomizes the bits in a controllable way, reversing the scrambling process. This LFSR descrambler implementation uses the Communications System Toolbox Descrambler block. From the

LFSR, the descrambled audio bits pass to the Bit to Integer Converter block. This block packs the descrambled 1-bit samples into 16-bit audio samples. The audio samples pass to the Data Type Conversion block, which converts the integer-based audio samples back into fixed-point samples.

In Simulink, the sfix16_En15 data type represents a signed (s) fixed-point (fix) signal with word length 16 and fraction length 15. Therefore, this model represents audio signals using the sfix16_En15 data type, except when converting to and from 1-bit binary signals. All 1-bit signals are represented by ufix1, as seen at the output of the Integer to Bit Converter block. The audio source has a frame size (or number of samples per frame) of 1024. For more information on fixed-point signals, please refer to Fixed-Point Numbers in the Simulink documentation.



Running the Model

You must run the example before you can listen to any of the audio signals.

You can run the example by clicking **Simulation > Run**.

You can hear the audio signals by clicking the model's yellow, audio icons.

Converter Block Settings

In the audio scrambler and descrambler subsystems, the Integer to Bit Converter block settings are:

- **Number of bits per integer:** `16`
- **Treat input values as:** `Signed`
- **Output bit order:** `MSB first`
- **Output data type:** `Inherit via internal rule`

In the audio scrambler and descrambler subsystems, the Bit to Integer Converter block settings are:

- **Number of bits per integer:** `16`
- **Input bit order:** `MSB first`
- **After bit packing, treat resulting integer values as:** `Signed`
- **Output data type:** `Inherit via internal rule`

Pair Block

Bit to Integer Converter

See Also

`de2bi` and `dec2bin`

Purpose

Integrate discrete-time signal, resetting to zero periodically

Library

Comm Filters

Description

The Integrate and Dump block creates a cumulative sum of the discrete-time input signal, while resetting the sum to zero according to a fixed schedule. When the simulation begins, the block discards the number of samples specified in the **Offset** parameter. After this initial period, the block sums the input signal along columns and resets the sum to zero every N input samples, where N is the **Integration period** parameter value. The reset occurs after the block produces its output at that time step.

Receiver models often use the integrate-and-dump operation when the system's transmitter uses a simple square-pulse model. Fiber optics and in spread-spectrum communication systems, such as CDMA (code division multiple access) applications, also use the operation.

This block accepts a scalar, column vector, or matrix input signal. When the input signal is not a scalar value, it must contain $k \cdot N$ rows for some positive integer k. For these input signals, the block processes each column independently.

Selecting **Output intermediate values** affects the contents, dimensions, and sample time as follows:

- If you clear the check box, then the block outputs the cumulative sum at each reset time.
  - If the input is a scalar value, then the output sample time is N times the input sample time and the block experiences a delay whose duration is one output sample period. In this case, the output dimensions match the input dimensions.
  - If the input is a $(k \cdot N)$-by-n matrix, then the output is k-by-n. In this case, the block experiences no delay and the output period matches the input period.

- If you select the check box, then the block outputs the cumulative sum at each time step. The output has the same sample time and the same matrix dimensions as the input.

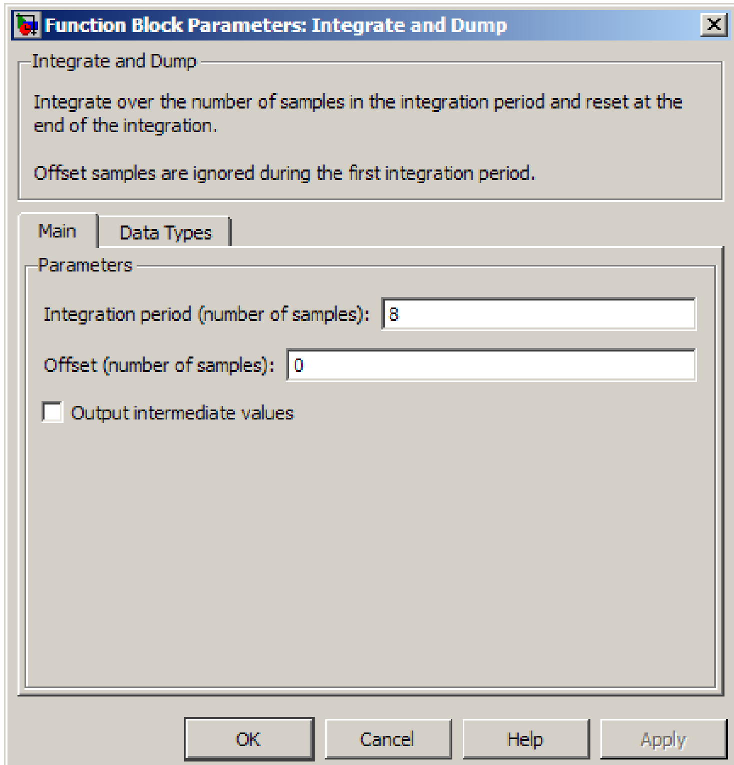
Transients and Delays

A nonzero value in the **Offset** parameter causes the block to output one or more zeros during the initial period while it discards input samples. If the input is a matrix with *n* columns and the **Offset** parameter is a length-*n* vector, then the m^{th} element of the **Offset** vector is the offset for the m^{th} column of data. If **Offset** is a scalar, then the block applies the same offset to each column of data. The output of initial zeros due to a nonzero **Offset** value is a transient effect, not a persistent delay.

When you clear **Output intermediate values**, the block's output is delayed, relative to its input, throughout the simulation:

- If the input is a scalar value, then the output is delayed by one sample after any transient effect is over. That is, after removing transients from the input and output, you can see the result of the m^{th} integration period in the output sample indexed by m+1.
- If the input is a column vector or matrix and the **Offset** parameter is nonzero, then after the transient effect is over, the result of each integration period appears in the output frame corresponding to the *last* input sample of that integration period. This is one frame later than the output frame corresponding to the first input sample of that integration period, in cases where an integration period spans two input frames. For an example of this situation, see "Example of Transient and Delay" on page 2-452.

Dialog Box



Integration period

The number of input samples between resets.

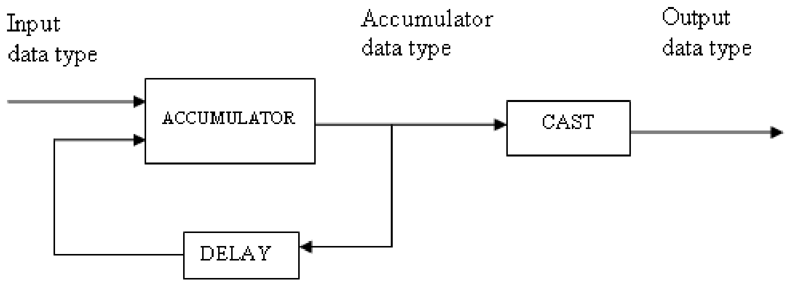
Offset

A nonnegative integer vector or scalar specifying the number of input samples to discard from each column of input data at the beginning of the simulation.

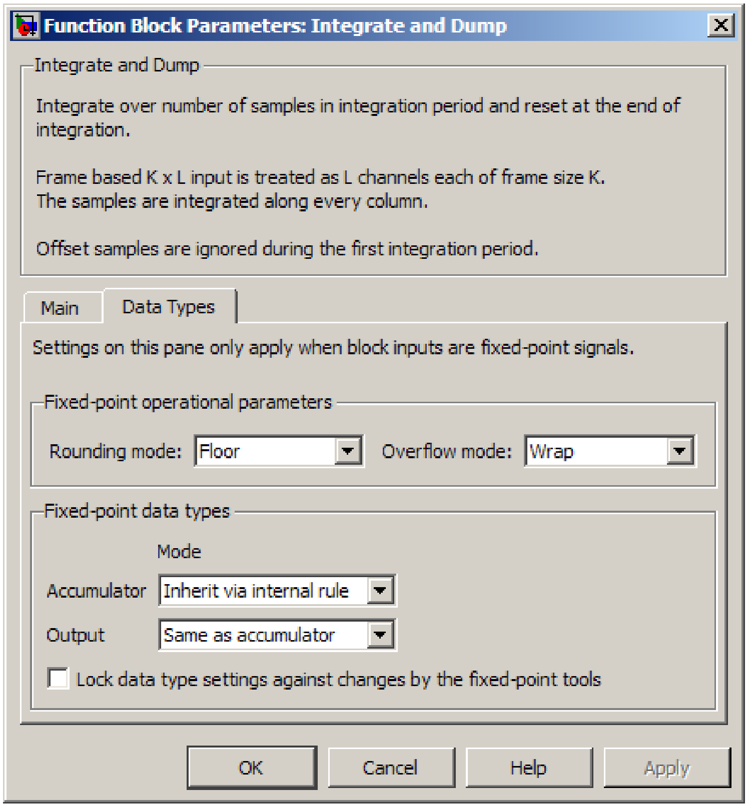
Output intermediate values

Determines whether the block outputs the intermediate cumulative sums between successive resets.

Fixed-Point Signal Flow Diagram



Fixed-Point Attributes



The settings for the following parameters only apply when block inputs are fixed-point signals.

Rounding mode

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

For more information, see "Rounding Modes" in the DSP System Toolbox documentation or "Rounding Mode: Simplest" in the Fixed-Point Designer documentation.

Overflow mode

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

- `Saturate` represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used.
- `Wrap` uses modulo arithmetic to cast an overflow back into the representable range of the data type. See "Modulo Arithmetic" for more information.

Accumulator—Mode

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

- When you select `Inherit via internal rule`, the block automatically calculates the accumulator 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 are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator.

Output

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

- 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`, enter the word length and the fraction length of the output, in bits.
- When you select `Slope and bias scaling`, enter the word length, in bits, and the slope of the output.

For additional information about the parameters pertaining to fixed-point applications, see “Specify Fixed-Point Attributes for Blocks”.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Fixed-point

Examples

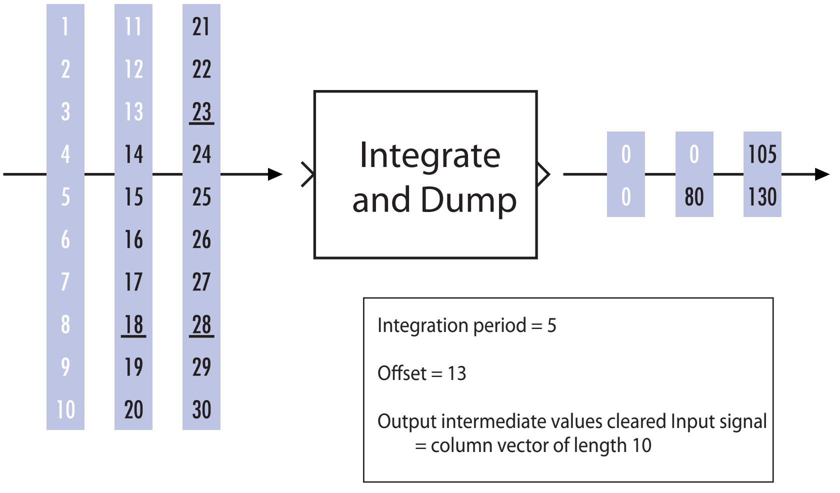
If **Integration period** is `4` and **Offset** is the scalar `3`, then the table below shows how the block treats the beginning of a ramp (`1`, `2`, `3`, `4`,...) in several situations. (The values shown in the table do not reflect vector sizes but merely indicate numerical values.)

Output intermediate values Check Box	**Input Signal Properties**	**First Several Output Values**
Cleared	Scalar	0, 0, 4+5+6+7, and 8+9+10+11, where one 0 is an initial transient value and the other 0 is a delay value that results from the cleared check box and scalar value input.
Cleared	Column vector of length 4	0, 4+5+6+7, and 8+9+10+11, where 0 is an initial delay value that results from the nonzero offset. The output is a scalar value.
Selected	Scalar	0, 0, 0, 4, 4+5, 4+5+6, 4+5+6+7, 8, 8+9, 8+9+10, 8+9+10+11, and 12, where the three 0s are initial transient values.
Selected	Column vector of length 4	0, 0, 0, 4, 4+5, 4+5+6, 4+5+6+7, 8, 8+9, 8+9+10, 8+9+10+11, and 12, where the three 0s are initial transient values. The output is a column vector of length 4.

In all cases, the block discards the first three input samples (1, 2, and 3).

Example of Transient and Delay

The figure below illustrates a situation in which the block exhibits both a transient effect for three output samples, as well as a one-sample delay in alternate subsequent output samples for the rest of the simulation. The figure also indicates how the input and output values are organized as column vectors. In each vector in the figure, the last sample of each integration period is underlined, discarded input samples are white, and transient zeros in the output are white.



The transient effect lasts for `ceil(13/5)` output samples because the block discards 13 input samples and the integration period is 5. The first output sample after the transient effect is over, 80, corresponds to the sum 14+15+16+17+18 and appears at the time of the input sample 18. The next output sample, 105, corresponds to the sum 19+20+21+22+23 and appears at the time of the input sample 23. Notice that the input sample 23 is one frame later than the input sample 19; that is, this five-sample integration period spans two input frames. As a result, the output of 105 is delayed compared to the first input (19) that contributes to that sum.

See Also

Windowed Integrator, Discrete-Time Integrator (Simulink documentation), Ideal Rectangular Pulse Filter

Interlacer

Purpose

Alternately select elements from two input vectors to generate output vector

Library

Sequence Operations

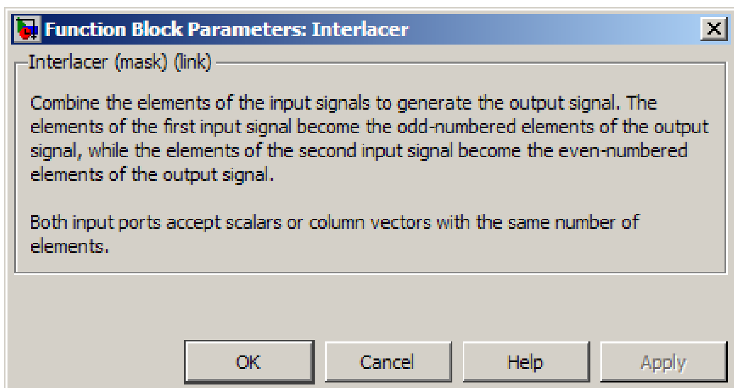
Description

The Interlacer block accepts two inputs that have the same vector size, complexity, and sample time. It produces one output vector by alternating elements from the first input (labeled `O` for odd) and from the second input (labeled `E` for even) . As a result, the output vector size is twice that of either input. The output vector has the same complexity and sample time of the inputs.

Both input ports accept scalars or column vectors with the same number of elements. The block accepts the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

This block can be useful for combining in-phase and quadrature information from separate vectors into a single vector.

Dialog Box



Examples

If the two input vectors have the values `[1; 2; 3; 4]` and `[5; 6; 7; 8]`, then the output vector is `[1; 5; 2; 6; 3; 7; 4; 8]`.

Pair Block

Deinterlacer

See Also

General Block Interleaver; Mux (Simulink documentation)

I/Q Imbalance

Purpose

Create complex baseband model of signal impairments caused by imbalances between in-phase and quadrature receiver components

Library

RF Impairments

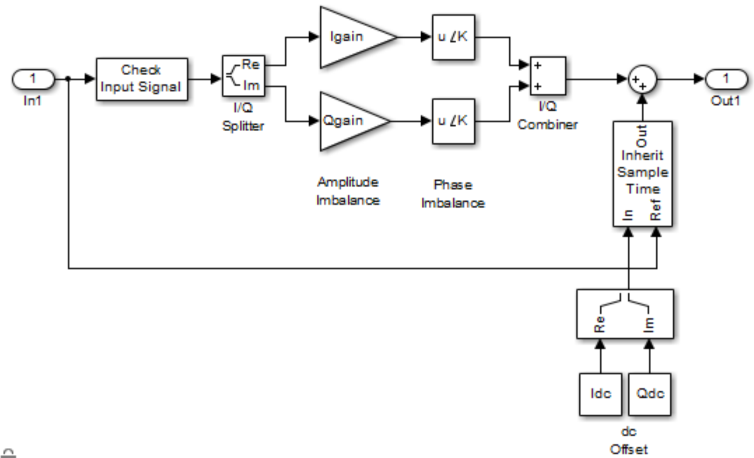
Description

The I/Q Imbalance block creates a complex baseband model of the signal impairments caused by imbalances between in-phase and quadrature receiver components. Typically, these are caused by differences in the physical channels for the two components of the signal.

The I/Q Imbalance block applies amplitude and phase imbalances to the in-phase and quadrature components of the input signal, and then combines the results into a complex signal. The block

1. Separates the signal into its in-phase and quadrature components.
2. Applies amplitude and phase imbalances, specified by the **I/Q amplitude imbalance (dB)** and **I/Q phase imbalance (deg)** parameters, respectively, to both components.
3. Combines the in-phase and quadrature components into a complex signal.
4. Applies an in-phase dc offset, specified by the **I dc offset** parameter, and a quadrature offset, specified by the **Q dc offset** parameter, to the signal.

The block performs these operations in the subsystem shown in the following diagram, which you can view by right-clicking the block and selecting **Mask** > **Look under mask**:



Let

I_a = I/Q amplitude imbalance

I_p = I/Q phase imbalance

I_{DC} = in-phase DC offset

Q_{DC} = quadrature DC offset

Also let $x = x_r + j * x_i$ be the complex input to the block, with x_r and x_i being the real and imaginary parts, respectively, of x. Let y be the complex output of the block.

Then, for an I/Q amplitude imbalance, I_a

$$y_{AmplitudeImbalance} = [10^{(0.5*\frac{I_a}{20})} * x_r] + j[10^{(-0.5*\frac{I_a}{20})} * x_i]$$

$$\square \; y_{rAmplitudeImbalance} + j*y_{iAmplitudeImbalance}$$

For an I/Q phase imbalance I_p

$$y_{PhaseImbalance} =$$

$$[\exp(-0.5 * j * \pi * \frac{I_p}{180}) * y_{r_{Amplitude\,\mathrm{Im}\,balance}}] + \{\exp[j(\frac{\pi}{2} + 0.5 * \pi * \frac{I_p}{180})] * y_{i_{Amplitude\,\mathrm{Im}\,balance}}\}$$
$$\triangleq y_{r_{PhaseImbalance}} + j * y_{i_{PhaseImbalance}}$$

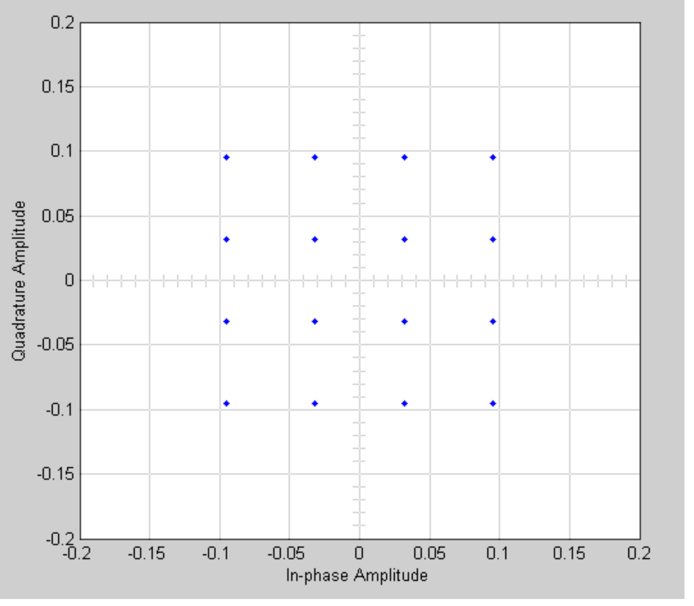
For DC offsets I_{DC} and Q_{DC}

$$y = (y_{r\,\mathrm{PhaseImbalance}} + I_{DC}) + j * (y_{i_{PhaseImbalance}} + Q_{DC})$$

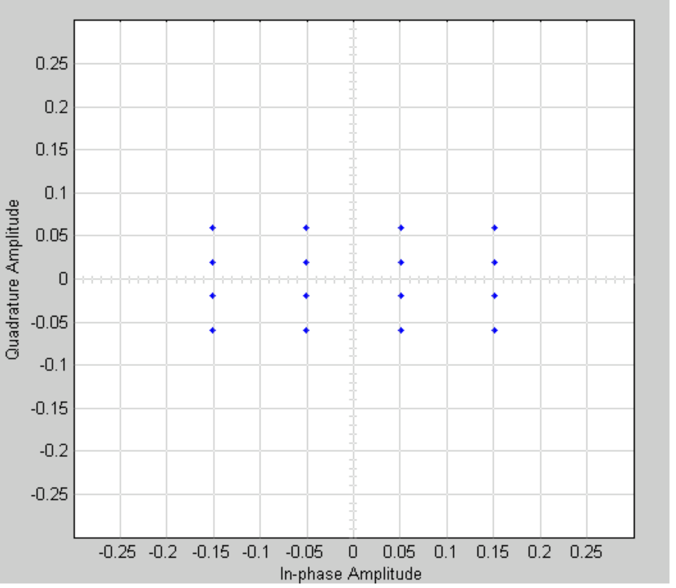
The value of the **I/Q amplitude imbalance (dB)** parameter is divided between the in-phase and quadrature components:

- If you enter a positive value *X* for the **I/Q amplitude imbalance (dB)**, the block applies a gain of +*X/2* dB to the in-phase component and a gain of -*X/2* dB to the quadrature component.
- If you enter a negative value *X* for the **I/Q amplitude imbalance (dB)**, the block applies a gain of -*X/2* dB to the in-phase component and a gain of +*X/2* dB to the quadrature component.

The effects of changing the block's parameters are illustrated by the following scatter plots of a signal modulated by 16-ary quadrature amplitude modulation (QAM) with an average power of `0.01` watts. The usual 16-ary QAM constellation without distortion is shown in the first scatter plot:

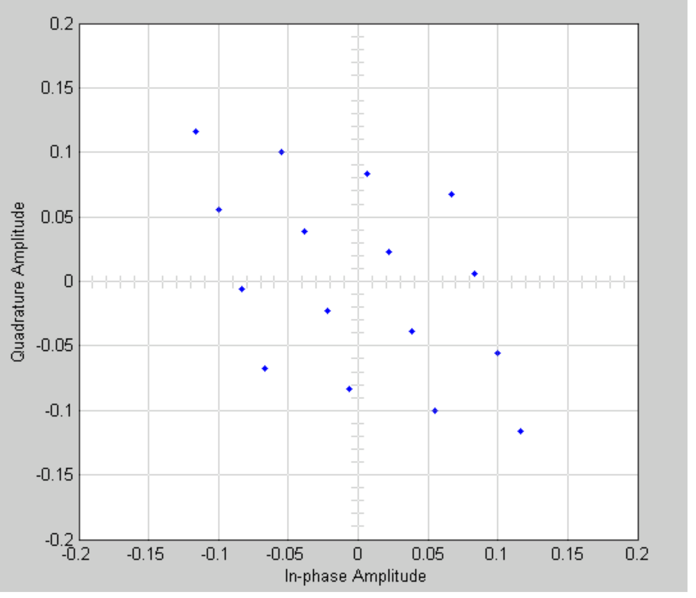


The following figure shows a scatter plot of an output signal, modulated by 16-ary QAM, from the I/Q block with **I/Q amplitude imbalance (dB)** set to `8` and all other parameters set to `0`:

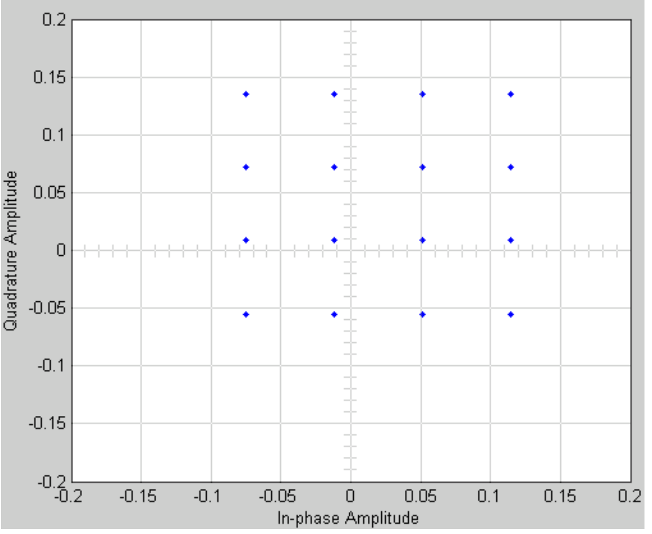


Observe that the scatter plot is stretched horizontally and compressed vertically compared to the undistorted constellation.

If you set **IQ phase imbalance (deg)** to `30` and all other parameters to `0`, the scatter plot is skewed clockwise by 30 degrees, as shown below:

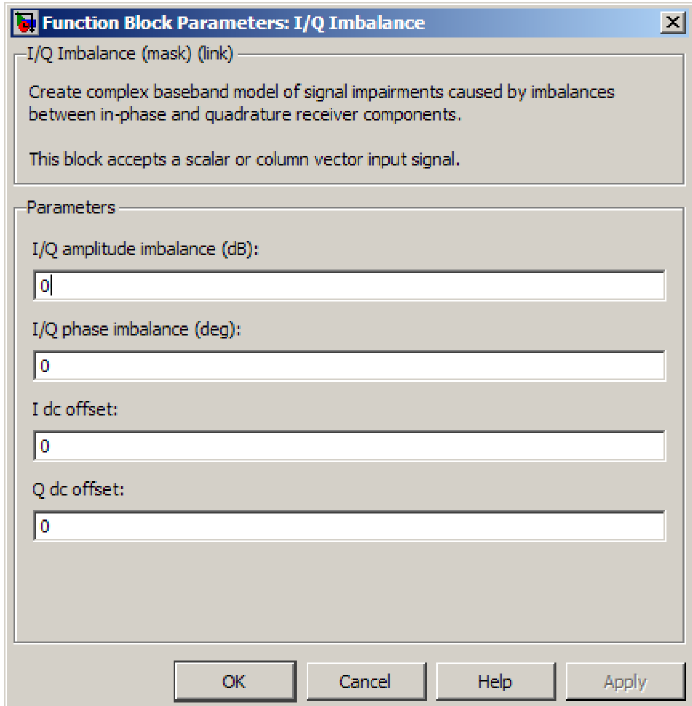


Setting the **I dc offset** to `0.02` and the **Q dc offset** to `0.04` shifts the constellation 0.02 to the right and 0.04 up, as shown below:



See "Illustrate RF Impairments That Distort a Signal" for a description of the model that generates this plot.

Dialog Box



I/Q amplitude imbalance (dB)
: Scalar specifying the I/Q amplitude imbalance in decibels.

I/Q phase imbalance (deg)
: Scalar specifying the I/Q phase imbalance in degrees.

I dc offset
: Scalar specifying the in-phase dc offset.

Q dc offset
: Scalar specifying the amplitude dc offset.

See Also

Memoryless Nonlinearity

Kasami Sequence Generator

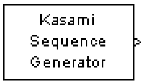
Purpose

Generate Kasami sequence from set of Kasami sequences

Library

Sequence Generators sublibrary of Comm Sources

Description



The Kasami Sequence Generator block generates a sequence from the set of Kasami sequences. The Kasami sequences are a set of sequences that have good cross-correlation properties.

This block can output sequences that vary in length during simulation. For more information about variable-size signals, see “Variable-Size Signal Basics” in the Simulink documentation.

Kasami Sequences

There are two sets of Kasami sequences: the *small set* and the *large set*. The large set contains all the sequences in the small set. Only the small set is optimal in the sense of matching Welch’s lower bound for correlation functions.

Kasami sequences have period $N = 2^n - 1$, where n is a nonnegative, even integer. Let u be a binary sequence of length N, and let w be the sequence obtained by decimating u by $2^{n/2} +1$. The small set of Kasami sequences is defined by the following formulas, in which T denotes the left shift operator, m is the shift parameter for w, and $\oplus$ denotes addition modulo 2.

$$K_s(u,n,m) = \begin{cases} u & m = -1 \\ u \oplus T^m w & m = 0,\ldots,2^{n/2} - 2 \end{cases}$$

Small Set of Kasami Sequences for n Even

Note that the small set contains $2^{n/2}$ sequences.

For mod(n, 4) = 2, the large set of Kasami sequences is defined as follows. Let v be the sequence formed by decimating the sequence u by $2^{n/2 + 1} + 1$. The large set is defined by the following table, in which k and m are the shift parameters for the sequences v and w, respectively.

$$K_L(u,n,k,m) = \begin{cases} u & k=-2;\ m=-1 \\ v & k=-1;\ m=-1 \\ u \oplus T^k v & k=0,\ldots,2^n-2;\ m=-1 \\ u \oplus T^m w & k=-2;\ m=0,\ldots,2^{n/2}-2 \\ v \oplus T^m w & k=-1;\ m=0,\ldots,2^{n/2}-2 \\ u \oplus T^k v \oplus T^m w & k=0,\ldots,2^n-2;\ m=0,\ldots,2^{n/2}-2 \end{cases}$$

Large Set of Kasami Sequences for mod(n, 4) = 2

The sequences described in the first three rows of the preceding figure correspond to the Gold sequences for mod(n, 4) = 2. See the reference page for the Gold Sequence Generator block for a description of Gold sequences. However, the Kasami sequences form a larger set than the Gold sequences.

The correlation functions for the sequences takes on the values

$\{-t(n),\ -s(n),\ -1,\ s(n)\ -2\ ,\ t(n)\ -\ 2\}$

where

$$t(n) = 1 + 2^{(n+2)/2},\ n \text{ even}$$
$$s(n) = \frac{1}{2}\left(t(n)+1\right)$$

Block Parameters

The **Generator polynomial** parameter specifies the generator polynomial, which determines the connections in the shift register that generates the sequence u. You can specify the **Generator polynomial** parameter using either of these formats:

- A vector that lists the coefficients of the polynomial in descending order of powers. The first and last entries must be 1. Note that the length of this vector is one more than the degree of the generator polynomial.

- A vector containing the exponents of z for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`.

For example, `[1 0 0 0 0 0 1 0 1]` and `[8 2 0]` represent the same polynomial, $p(z) = z^8+z^2+1$.

The **Initial states** parameter specifies the initial states of the shift register that generates the sequence u. **Initial States** is a binary scalar or row vector of length equal to the degree of the **Generator polynomial**. If you choose a binary scalar, the block expands the scalar to a row vector of length equal to the degree of the **Generator polynomial**, all of whose entries equal the scalar.

The **Sequence index** parameter specifies the shifts of the sequences v and w used to generate the output sequence. You can specify the parameter in either of two ways:

- To generate sequences from the small set, for n is even, you can specify the **Sequence index** as an integer m. The range of m is [-1, ..., $2^{n/2}$ - 2]. The following table describes the output sequences corresponding to **Sequence index** `m`:

Sequence Index	Range of Indices	Output Sequence
`-1`	m = -1	u
`m`	m = 0, ... , $2^{n/2}$ - 2	$u \oplus T^m w$

- To generate sequences from the large set, for mod (n, 4) = 2, where n is the degree of the **Generator polynomial**, you can specify **Sequence index** as an integer vector [k m]. In this case, the output sequence is from the large set. The range for k is [-2, ..., 2^n - 2], and the range for m is [-1, ..., $2^{n/2}$ - 2]. The following table describes the output sequences corresponding to **Sequence index** [k m]:

Sequence Index [k m]	Range of Indices	Output Sequence
`[-2 -1]`	k = -2, m = -1	u
`[-1 -1]`	k = -1, m = -1	v
`[k -1]`	$k = 0, 1, \ldots, 2^n - 2$ $m = -1$	$u \oplus T^k v$
`[-2 m]`	$k = -2$ $m = 0, 1, \ldots, 2^{n/2} - 2$	$u \oplus T^m w$
`[-1 m]`	$k = -1$ $m = 0, \ldots, 2^{n/2} - 2$	$v \oplus T^m w$
`[k m]`	$k = 0, \ldots, 2^n - 2$ $m = 0, \ldots, 2^{n/2} - 2$	$u \oplus T^k v \oplus T^m w$

You can shift the starting point of the Kasami sequence with the **Shift** parameter, which is an integer representing the length of the shift.

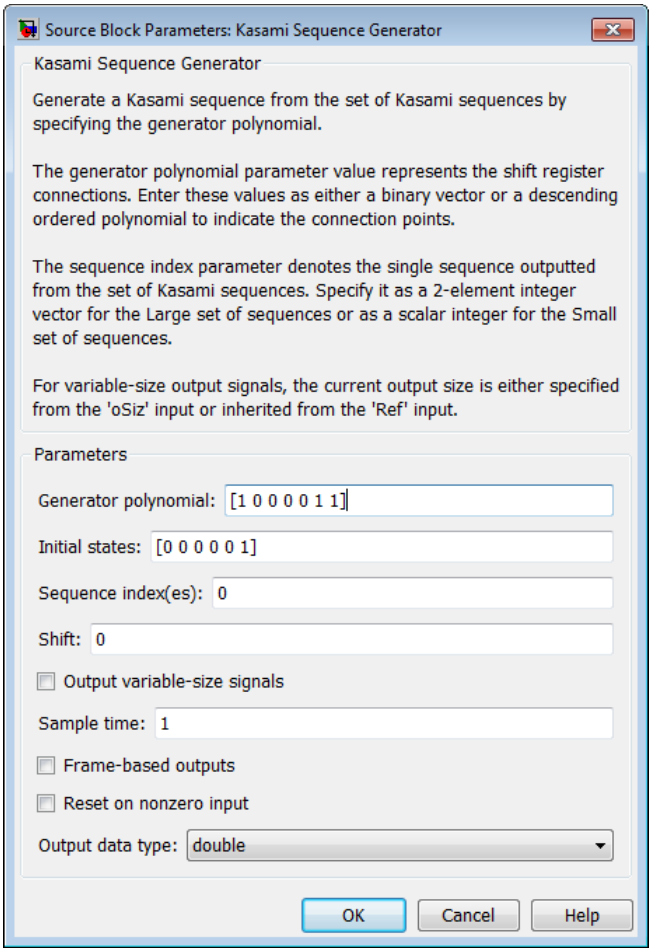
You can use an external signal to reset the values of the internal shift register to the initial state by selecting **Reset on nonzero input**. This creates an input port for the external signal in the Kasami Sequence Generator block. The way the block resets the internal shift register depends on whether its output signal and the reset signal are sample-based or frame-based. See "Example: Resetting a Signal" on page 2-723 for an example.

Polynomials for Generating Kasami Sequences

The following table lists some of the polynomials that you can use to generate the Kasami set of sequences.

n	N	Polynomial	Set
4	15	[4 1 0]	Small
6	63	[6 1 0]	Large

n	N	Polynomial	Set
8	255	[8 4 3 2 0]	Small
10	1023	[10 3 0]	Large
12	4095	[12 6 4 1 0]	Small



Dialog Box

Generator polynomial

Binary vector specifying the generator polynomial for the sequence u.

Initial states
Binary scalar or row vector of length equal to the degree of the **Generator polynomial**, which specifies the initial states of the shift register that generates the sequence *u*.

Sequence index
Integer or vector specifying the shifts of the sequences *v* and *w* used to generate the output sequence.

Shift
Integer scalar that determines the offset of the Kasami sequence from the initial time.

Output variable-size signals
Select this if you want the output sequences to vary in length during simulation. The default selection outputs fixed-length signals.

Maximum output size source
Specify how the block defines maximum output size for a signal.

- When you select `Dialog parameter`, the value you enter in the **Maximum output size** parameter specifies the maximum size of the output. When you make this selection, the `oSiz` input port specifies the current size of the output signal and the block output inherits sample time from the input signal. The input value must be less than or equal to the **Maximum output size** parameter.
- When you select `Inherit from reference port`, the block output inherits sample time, maximum size, and current size from the variable-sized signal at the Ref input port.

This parameter only appears when you select **Output variable-size signals**. The default selection is `Dialog parameter`.

Maximum output size
Specify a two-element row vector denoting the maximum output size for the block. The second element of the vector must be

1. For example, [10 1] gives a 10-by-1 maximum sized output signal. This parameter only appears when you select **Output variable-size signals**.

Sample time
Period of each element of the output signal.

Frame-based outputs
Determines whether the output is frame-based or sample-based.

Samples per frame
The number of samples in a frame-based output signal. This field is active only if you select **Frame-based outputs**.

Reset on nonzero input
When selected, you can specify an input signal that resets the internal shift registers to the original values of the **Initial states.**

Output data type
The output type of the block can be specified as a `boolean` or `double`. By default, the block sets this to `double`.

See Also

Gold Sequence Generator, PN Sequence Generator

Reference

[1] Peterson and Weldon, *Error Correcting Codes*, 2nd Ed., MIT Press, Cambridge, MA, 1972.

[2] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

[3] Sarwate, D. V. and Pursley, M.B., "Crosscorrelation Properties of Pseudorandom and Related Sequences," *Proc. IEEE*, Vol. 68, No. 5, May 1980, pp. 583-619.

LDPC Decoder

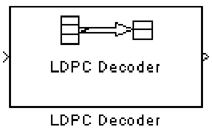
Purpose

Decode binary low-density parity-check code specified by parity-check matrix

Library

Block sublibrary of Error Detection and Correction

Description



This block implements the message-passing algorithm for decoding low-density parity-check (LDPC) codes, which are linear error control codes with sparse parity-check matrices and long block lengths that can attain performance near the Shannon limit.

The LDPC Decoder block is designed to:

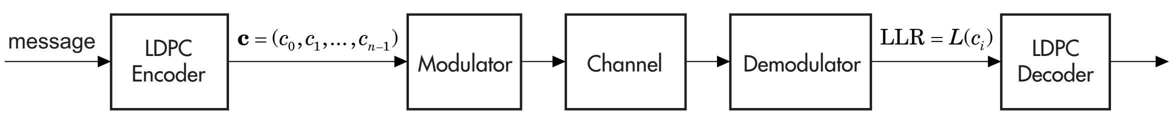
- Decode generic binary LDPC codes where no patterns in the parity-check matrix are assumed.
- Execute a number of iterations you specify or run until all parity-checks are satisfied.
- Output hard decisions or soft decisions (log-likelihood ratios) for decoded bits.

$(n-k)$ and n are the number of rows and columns, respectively, in the parity-check matrix.

This block accepts a real-valued, $n \times 1$ column vector input signal of type double. Each element is the log-likelihood ratio for a received bit (more likely to be 0 if the log-likelihood ratio is positive). The first k elements correspond to the information part of a codeword.

Both the input and the output are discrete-time signals. The ratio of the output sample time to the input sample time is n/k if only the information part is decoded, and 1 if the entire codeword is decoded.

Decoding Algorithm



The input to the LDPC decoder is the log-likelihood ratio (LLR), $L(c_i)$, which is defined by the following equation

$$L(c_i) = \log\left(\frac{\Pr(c_i = 0 \mid \text{channel output for } c_i)}{\Pr(c_i = 1 \mid \text{channel output for } c_i)}\right)$$

where c_i is the *i*th bit of the transmitted codeword, *c*. There are three key variables in the algorithm: $L(r_{ji})$, $L(q_{ij})$, and $L(Q_i)$. $L(q_{ij})$ is initialized as $L(q_{ij}) = L(c_i)$. For each iteration, update $L(r_{ji})$, $L(q_{ij})$, and $L(Q_i)$ using the following equations

$$L(r_{ji}) = 2\,\text{atanh}\left(\prod_{i' \in V_j \backslash i} \tanh\left(\frac{1}{2}L(q_{i'j})\right)\right)$$

$$L(q_{ij}) = L(c_i) + \sum_{j' \in C_i \backslash j} L(r_{j'i})$$

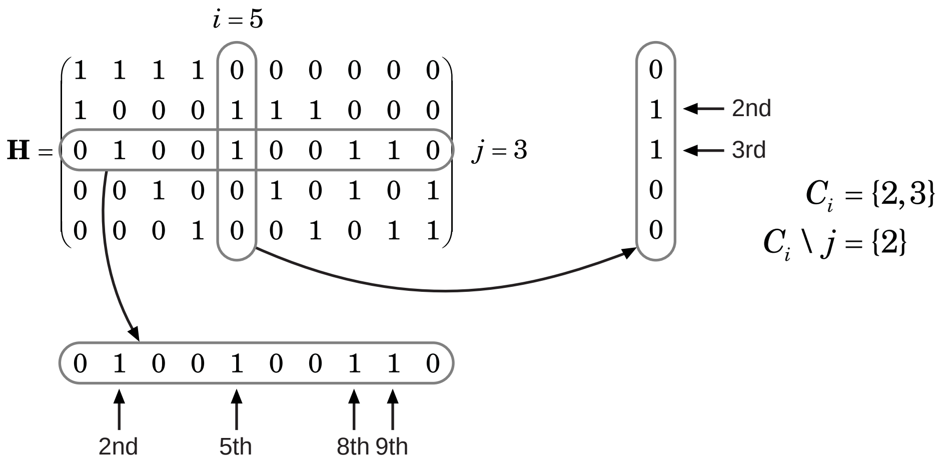
$$L(Q_i) = L(c_i) + \sum_{j' \in C_i} L(r_{j'i})$$

where the index sets, $C_i \backslash j$ and $V_j \backslash i$, are chosen as shown in the following example.

Suppose you have the following parity-check matrix **H**:

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

For $i = 5$ and $j = 3$, the index sets would be



$$V_j = \{2,5,8,9\}$$

$$V_j \setminus i = \{2,8,9\}$$

At the end of each iteration, $L(Q_i)$ provides an updated estimate of the *a posteriori* log-likelihood ratio for the transmitted bit c_i.

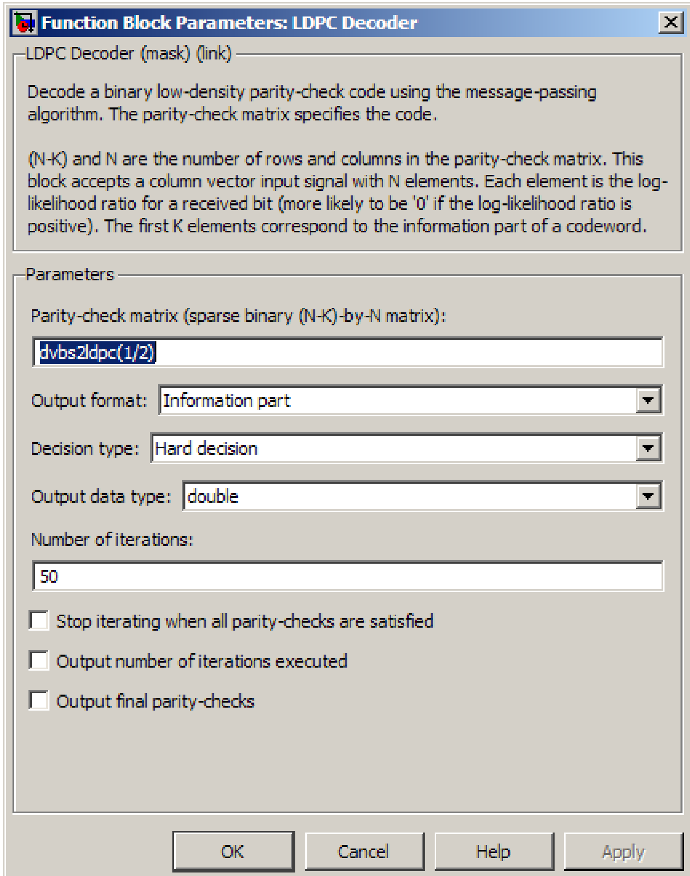
The soft-decision output for c_i is $L(Q_i)$. The hard-decision output for c_i is 1 if $L(Q_i) < 0$, and 0 otherwise.

If the property `DoParityCheck` is set to `'no'`, the algorithm iterates as many times as specified by the **Number of iterations** parameter.

If the property `DoParityCheck` is set to `'yes'`, then at the end of each iteration the algorithm verifies the parity check equation ($\mathbf{Hc}^{\mathrm{T}} = 0$) and stops if it is satisfied.

In this algorithm, atanh(1) and atanh(-1) are set to be 19.07 and -19.07 respectively to avoid infinite numbers from being used in the algorithm's equations. These numbers were chosen because MATLAB returns 1 for tanh(19.07) and -1 for tanh(-19.07), due to finite precision.

Dialog Box



Parity-check matrix

This parameter accepts a sparse matrix with dimension n -k by n (where n >k > 0) of real numbers. All nonzero elements must be equal to 1. The upper bound limit for the value of n is 2^{31}-1

Output format

The output is a real-valued column vector signal. The options are `Information part` and `Whole codeword`.

- When you this parameter to `Information part`, the output contains k elements.
- When you set this parameter to `whole codeword`, the output contains n elements

Decision type

The options are `Hard decision` and `Soft decision`.

- When you set this parameter to `Hard decision`, the output is decoded bits (of type double or boolean).
- When you set this parameter to `Soft decision`, the output is log-likelihood ratios (of type double).

Output data type

This parameter appears only when **Decision type** is set to `Hard decision`.

The options are `boolean` and `double`.

Number of iterations

This can be any positive integer.

Stop iterating when all parity checks are satisfied

If checked, the block will determine whether the parity checks are satisfied after each iteration and stop if all are satisfied.

Output number of iterations executed

Creates an output port on the block when selected.

Output final parity checks

Creates an output port on the block when selected.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point
Out	• Double-precision floating point • Boolean when **Decision type** is `Hard decision`

Examples

Enter `commdvbs2` at the command line to see an example that uses this block.

References

[1] Gallager, Robert G., *Low-Density Parity-Check Codes*, Cambridge, MA, MIT Press, 1963.

See Also

LDPC Encoder | `comm.LDPCDecoder` | `dvbs2ldpc`

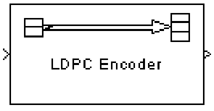
Purpose

Encode binary low-density parity-check code specified by parity-check matrix

Library

Block sublibrary of Error Detection and Correction

Description



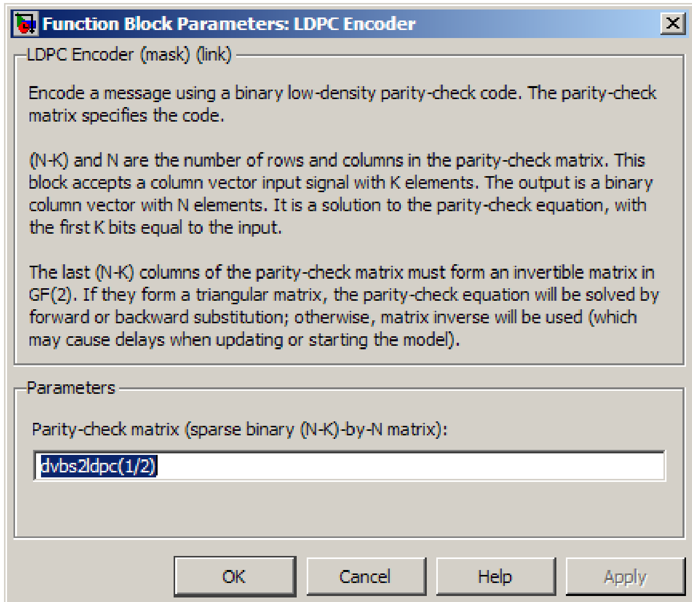
LDPC Encoder

This block supports encoding of low-density parity-check (LDPC) codes, which are linear error control codes with sparse parity-check matrices and long block lengths that can attain performance near the Shannon limit.

Both the input and the output are discrete-time signals. The ratio of the output sample time to the input sample time is k/n. The input must be a real $k \times 1$ column vector signal.

The output signal inherits the data type from the input signal, and the input must be binary-valued (0 or 1). For information about the data types each block port supports, see the "Supported Data Type" on page 2-480 table on this page.

Note Model initialization or update may take a long time, because a large matrix may need to be inverted (when the last $(n - k)$ columns of the parity-check matrix is not triangular).



Dialog Box

Parity-check matrix

This block can accept a sparse matrix with dimension n -k by n (where $n > k > 0$) of real numbers. All nonzero elements must be equal to 1. The upper bound limit for the value of n is 2^{31}-1

The default value is the parity-check matrix of the half-rate LDPC code from the DVB-S.2 standard.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Examples

Enter `commdvbs2` at the command line to see an example that uses this block.

See Also

LDPC Decoder | `comm.LDPCEncoder` | `dvbs2ldpc`

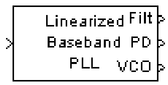
Purpose

Implement linearized version of baseband phase-locked loop

Library

Components sublibrary of Synchronization

Description



The Linearized Baseband PLL block is a feedback control system that automatically adjusts the phase of a locally generated signal to match the phase of an input signal. Unlike the Phase-Locked Loop block, this block uses a baseband model method. Unlike the Baseband PLL block, which uses a nonlinear model, this block simplifies the computations by using x to approximate $\sin(x)$. The baseband PLL model depends on the amplitude of the incoming signal but does not depend on a carrier frequency.

This PLL has these three components:

- An integrator used as a phase detector.
- A filter. You specify the filter's transfer function using the **Lowpass filter numerator** and **Lowpass filter denominator** parameters. Each is a vector that gives the respective polynomial's coefficients in order of descending powers of s.

  To design a filter, you can use functions such as `butter`, `cheby1`, and `cheby2` in Signal Processing Toolbox software. The default filter is a Chebyshev type II filter whose transfer function arises from the command below.

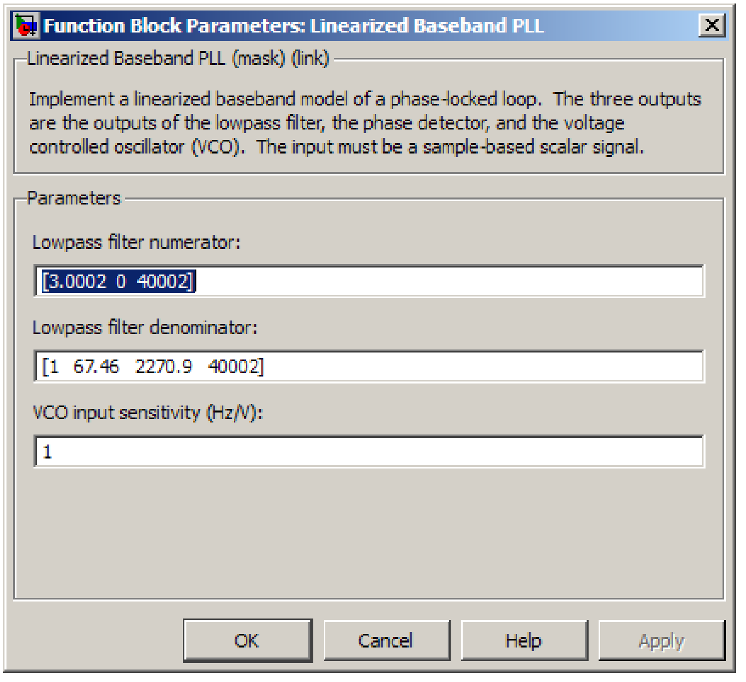
  ```
  [num, den] = cheby2(3,40,100,'s')
  ```
- A voltage-controlled oscillator (VCO). You specify the sensitivity of the VCO signal to its input using the **VCO input sensitivity** parameter. This parameter, measured in Hertz per volt, is a scale factor that determines how much the VCO shifts from its quiescent frequency.

This block accepts a sample-based scalar input signal. The input signal represents the received signal. The three output ports produce:

- The output of the filter

- The output of the phase detector
- The output of the VCO

Dialog Box



Lowpass filter numerator

The numerator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of s.

Lowpass filter denominator

The denominator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of s.

VCO input sensitivity (Hz/V)
This value scales the input to the VCO and, consequently, the shift from the VCO's quiescent frequency.

See Also

Baseband PLL, Phase-Locked Loop

References

For more information about phase-locked loops, see the works listed in "Selected Bibliography for Synchronization" in *Communications System Toolbox User's Guide*.

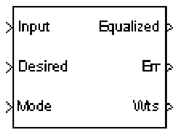
Purpose

Equalize using decision feedback equalizer that updates weights with LMS algorithm

Library

Equalizers

Description



The LMS Decision Feedback Equalizer block uses a decision feedback equalizer and the LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the LMS algorithm to update the weights, once per symbol. If the **Number of samples per symbol** parameter is `1`, then the block implements a symbol-spaced equalizer; otherwise, the block implements a fractionally spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of forward taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input, as described in "Reference Signal and Operation Modes" in *Communications System Toolbox User's Guide*.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output, as described in "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Using Adaptive Equalizers" in *Communications System Toolbox User's Guide*.

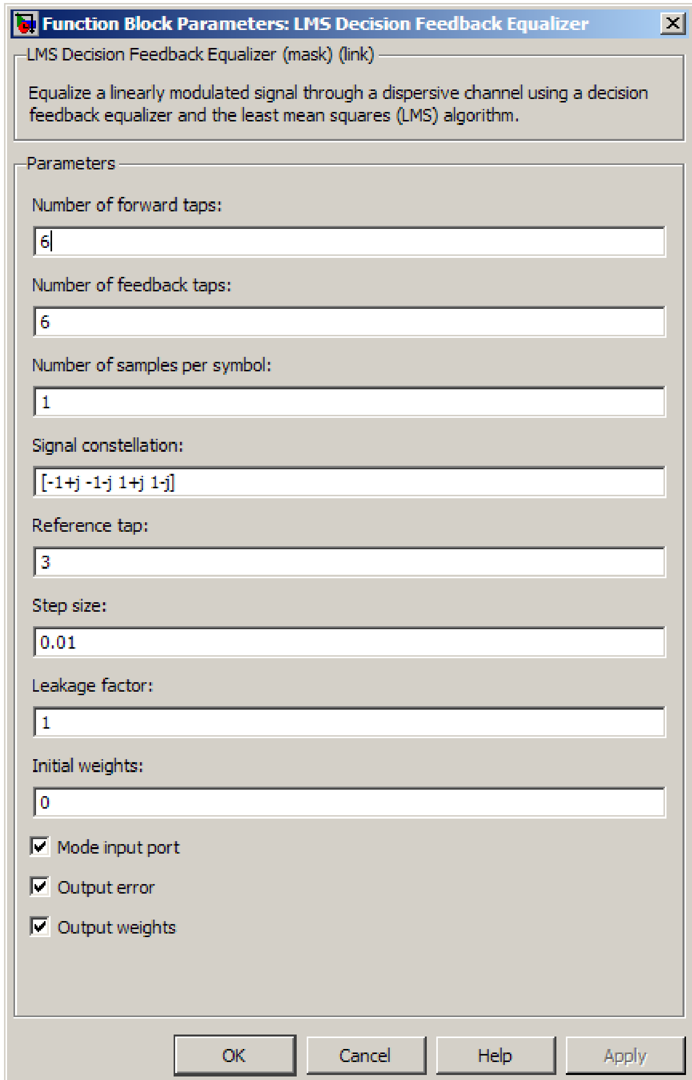
Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap of the forward filter.

Dialog Box



Number of forward taps

The number of taps in the forward filter of the decision feedback equalizer.

Number of feedback taps
The number of taps in the feedback filter of the decision feedback equalizer.

Number of samples per symbol
The number of input samples for each symbol.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulation.

Reference tap
A positive integer less than or equal to the number of forward taps in the equalizer.

Step size
The step size of the LMS algorithm.

Leakage factor
The leakage factor of the LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights
A vector that concatenates the initial weights for the forward and feedback taps.

Mode input port
If you select this check box, the block has an input port that enables you to toggle between training and decision-directed mode. For training, the mode input must be 1, and for decision directed, the mode must be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error
If you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

If you select this check box, the block outputs the current forward and feedback weights, concatenated into one vector.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

See Also

LMS Linear Equalizer, Normalized LMS Decision Feedback Equalizer, Sign LMS Decision Feedback Equalizer, Variable Step LMS Decision Feedback Equalizer, RLS Decision Feedback Equalizer, CMA Equalizer

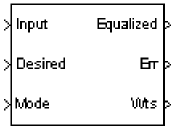
Purpose

Equalize using linear equalizer that meditorsupdates weights with LMS algorithm

Library

Equalizers

Description



The LMS Linear Equalizer block uses a linear equalizer and the LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced (i.e. T-spaced) equalizer. When you set the **Number of samples per symbol** parameter to a value greater than one, the block updates the weights once every N^{th} sample for a T/N-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of taps** parameter.

The `Equalized` port outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input, as described in "Reference Signal and Operation Modes" in *Communications System Toolbox User's Guide*.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output, as described in "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Decision-Directed Mode and Training Mode

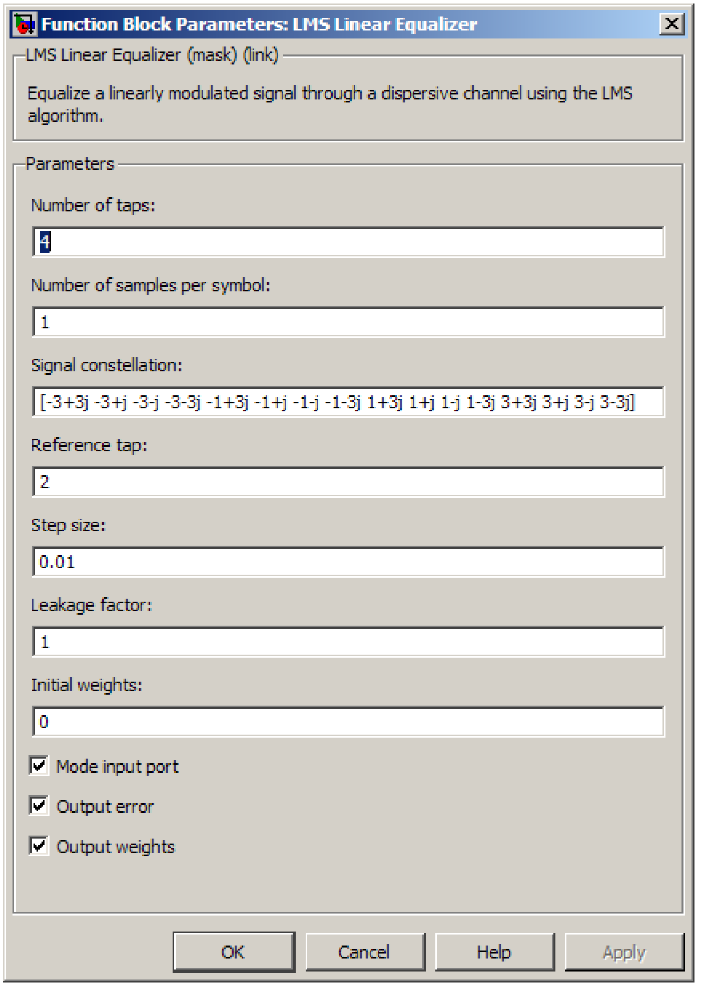
To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Using Adaptive Equalizers" in *Communications System Toolbox User's Guide*.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap.



Dialog Box

Number of taps

The number of taps in the filter of the linear equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulated signal, as determined by the modulator in your model

Reference tap
A positive integer less than or equal to the number of taps in the equalizer.

Step size
The step size of the LMS algorithm.

Leakage factor
The leakage factor of the LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights
A vector that lists the initial weights for the taps.

Mode input port
If you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, and for decision directed, the mode must be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error
If you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights
If you select this check box, the block outputs the current weights.

Examples

See "Implement LMS Linear Equalizer Using Simulink" for an example that uses this block.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

See Also

LMS Decision Feedback Equalizer, Normalized LMS Linear Equalizer, Sign LMS Linear Equalizer, Variable Step LMS Linear Equalizer, RLS Linear Equalizer, CMA Equalizer

Matrix Deinterleaver

Purpose

Permute input symbols by filling matrix by columns and emptying it by rows

Library

Block sublibrary of Interleaving

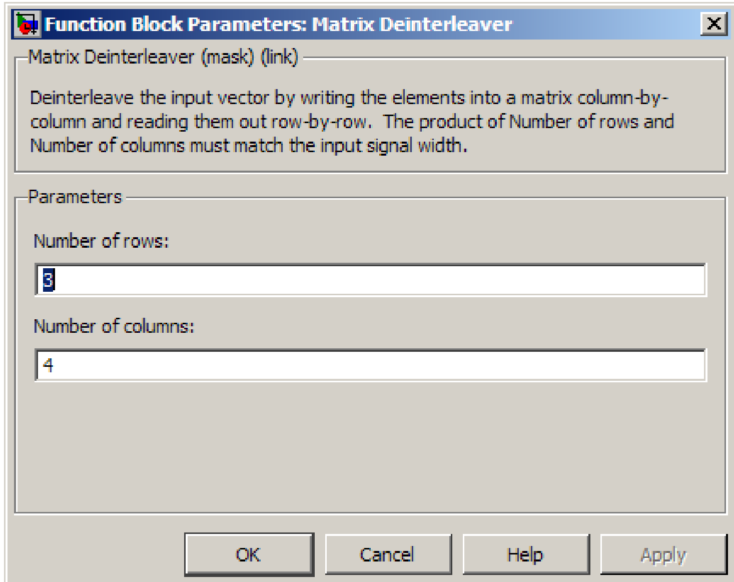
Description

The Matrix Deinterleaver block performs block deinterleaving by filling a matrix with the input symbols column by column and then sending the matrix contents to the output port row by row. The **Number of rows** and **Number of columns** parameters are the dimensions of the matrix that the block uses internally for its computations.

This block accepts a column vector input signal. The length of the input vector must be **Number of rows** times **Number of columns**.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

Dialog Box



Number of rows
: The number of rows in the matrix that the block uses for its computations.

Number of columns
: The number of columns in the matrix that the block uses for its computations.

Examples

If the **Number of rows** and **Number of columns** parameters are 2 and 3, respectively, then the deinterleaver uses a 2-by-3 matrix for its internal computations. Given an input signal of `[1; 2; 3; 4; 5; 6]`, the block produces an output of `[1; 3; 5; 2; 4; 6]`.

Pair Block

Matrix Interleaver

See Also

General Block Deinterleaver

Matrix Helical Scan Deinterleaver

Purpose

Restore ordering of input symbols by filling matrix along diagonals

Library

Block sublibrary of Interleaving

Description

The Matrix Helical Scan Deinterleaver block performs block deinterleaving by filling a matrix with the input symbols in a helical fashion and then sending the matrix contents to the output port row by row. The **Number of rows** and **Number of columns** parameters are the dimensions of the matrix that the block uses internally for its computations.

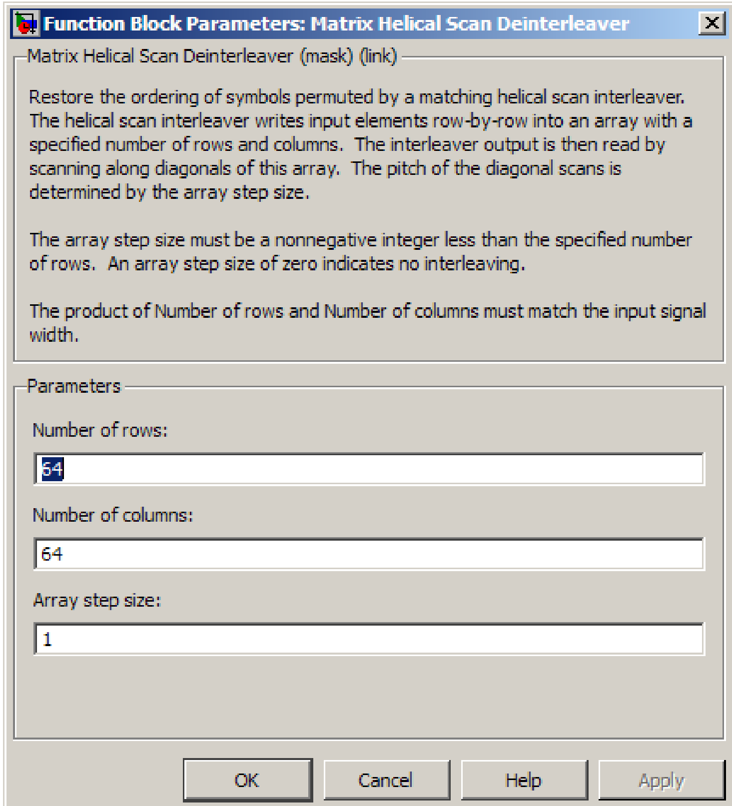
Helical fashion means that the block places input symbols along diagonals of the matrix. The number of elements in each diagonal matches the **Number of columns** parameter, after the block wraps past the edges of the matrix when necessary. The block traverses diagonals so that the row index and column index both increase. Each diagonal after the first one begins one row below the first element of the previous diagonal.

The **Array step size** parameter is the slope of each diagonal, that is, the amount by which the row index increases as the column index increases by one. This parameter must be an integer between zero and the **Number of rows** parameter. If the **Array step size** parameter is zero, then the block does not deinterleave and the output is the same as the input.

This block accepts a column vector input signal. The number of elements of the input vector must be the product of **Number of rows** and **Number of columns**.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

Dialog Box



Number of rows

The number of rows in the matrix that the block uses for its computations.

Number of columns

The number of columns in the matrix that the block uses for its computations.

Array step size

The slope of the diagonals that the block writes.

Pair Block Matrix Helical Scan Interleaver

See Also General Block Deinterleaver

Purpose

Permute input symbols by selecting matrix elements along diagonals

Library

Block sublibrary of Interleaving

Description

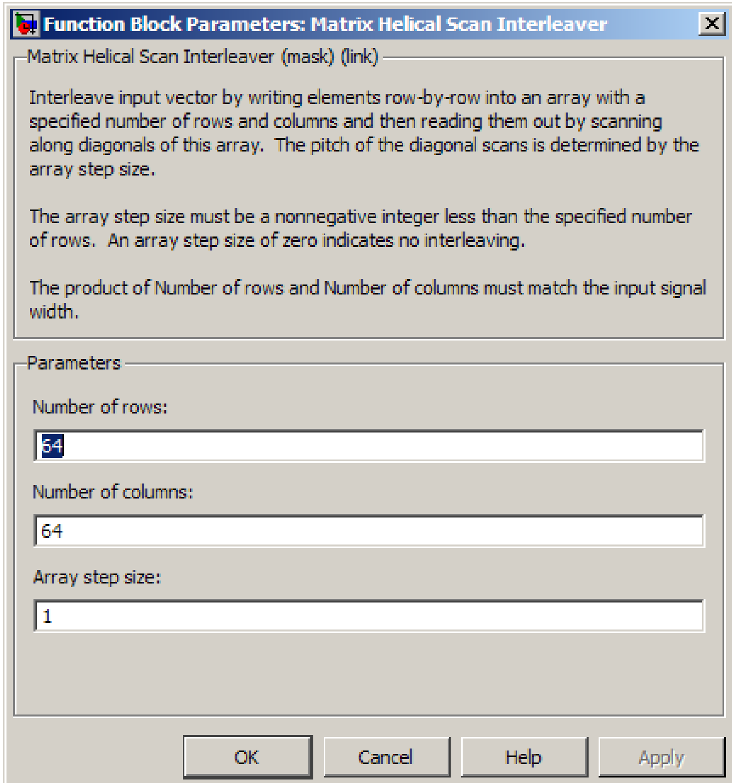
The Matrix Helical Scan Interleaver block performs block interleaving by filling a matrix with the input symbols row by row and then sending the matrix contents to the output port in a helical fashion. The **Number of rows** and **Number of columns** parameters are the dimensions of the matrix that the block uses internally for its computations.

Helical fashion means that the block selects output symbols by selecting elements along diagonals of the matrix. The number of elements in each diagonal matches the **Number of columns** parameter, after the block wraps past the edges of the matrix when necessary. The block traverses diagonals so that the row index and column index both increase. Each diagonal after the first one begins one row below the first element of the previous diagonal.

The **Array step size** parameter is the slope of each diagonal, that is, the amount by which the row index increases as the column index increases by one. This parameter must be an integer between zero and the **Number of rows** parameter. If the **Array step size** parameter is zero, then the block does not interleave and the output is the same as the input.

This block accepts a column vector input signal. The number of elements of the input vector must be the product of **Number of rows** and **Number of columns**.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.



Dialog Box

Number of rows
: The number of rows in the matrix that the block uses for its computations.

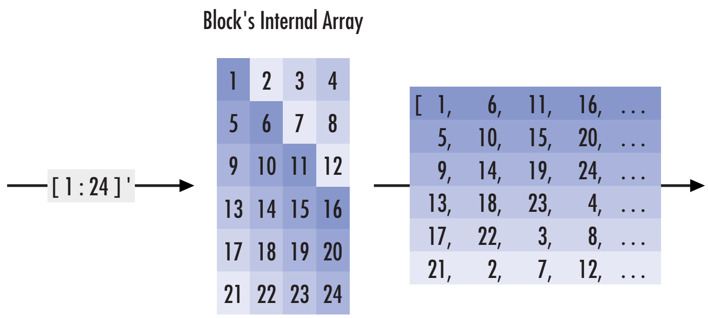
Number of columns
: The number of columns in the matrix that the block uses for its computations.

Array step size
: The slope of the diagonals that the block reads.

Examples

If the **Number of rows** and **Number of columns** parameters are 6 and 4, respectively, then the interleaver uses a 6-by-4 matrix for its internal computations. If the **Array step size** parameter is 1, then the diagonals are as shown in the figure below. Positions with the same color form part of the same diagonal, and diagonals with darker colors precede those with lighter colors in the output signal.

Given an input signal of `[1:24]'`, the block produces an output of




```
[1; 6; 11; 16; 5; 10; 15; 20; 9; 14; 19; 24; 13; 18; 23;...
4; 17; 22; 3; 8; 21; 2; 7; 12]
```

Pair Block

Matrix Helical Scan Deinterleaver

See Also

General Block Interleaver

Purpose

Permute input symbols by filling matrix by rows and emptying it by columns

Library

Block sublibrary of Interleaving

Description

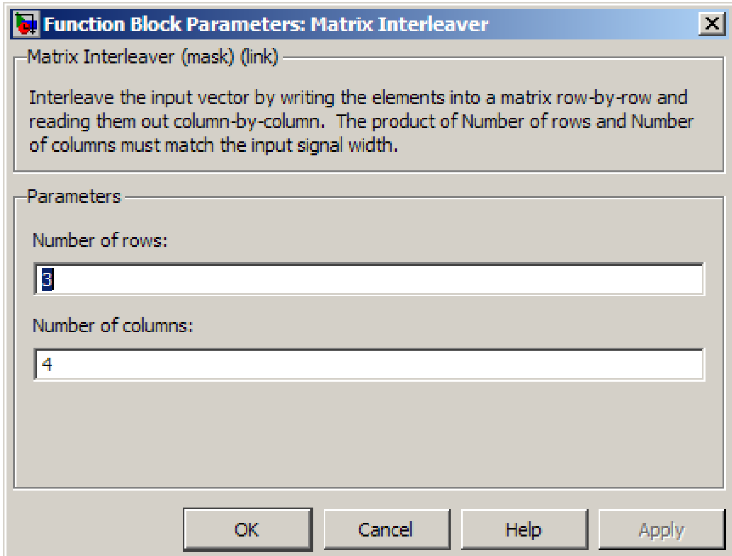
The Matrix Interleaver block performs block interleaving by filling a matrix with the input symbols row by row and then sending the matrix contents to the output port column by column.

The **Number of rows** and **Number of columns** parameters are the dimensions of the matrix that the block uses internally for its computations.

This block accepts a column vector input signal. The number of elements of the input vector must be the product of **Number of rows** and **Number of columns**.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

Dialog Box



Number of rows

The number of rows in the matrix that the block uses for its computations.

Number of columns

The number of columns in the matrix that the block uses for its computations.

Examples

If the **Number of rows** and **Number of columns** parameters are 2 and 3, respectively, then the interleaver uses a 2-by-3 matrix for its internal computations. Given an input signal of `[1; 2; 3; 4; 5; 6]`, the block produces an output of `[1; 4; 2; 5; 3; 6]`.

Pair Block

Matrix Deinterleaver

See Also

General Block Interleaver

M-DPSK Demodulator Baseband

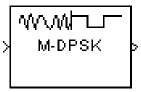
Purpose

Demodulate DPSK-modulated data

Library

PM, in Digital Baseband sublibrary of Modulation

Description



The M-DPSK Demodulator Baseband block demodulates a signal that was modulated using the M-ary differential phase shift keying method. The input is a baseband representation of the modulated signal. The input and output for this block are discrete-time signals. This block accepts a scalar-valued or column vector input signal. For information about the data types each block port supports, see the “Supported Data Types” on page 2-506 table on this page.

The **M-ary number** parameter, M, is the number of possible output symbols that can immediately follow a given output symbol. The block compares the current symbol to the previous symbol. The block’s first output is the initial condition of zero (or a group of zeros, if the **Output type** parameter is set to `Bit`) because there is no previous symbol.

Integer-Valued Signals and Binary-Valued Signals

If you set the **Output type** parameter to `Integer`, then the block demodulates a phase difference of

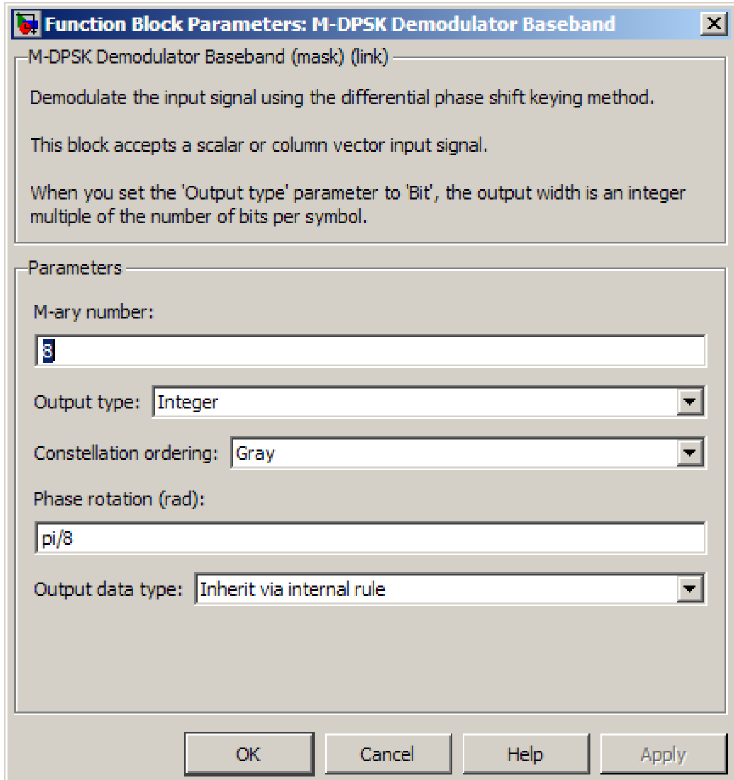
$$\theta + 2\pi k/M$$

to k, where θ represents the **Phase rotation** parameter and k represents an integer between `0` and M-1.

When you set the **Output type** parameter to `Bit`, the block outputs binary-valued signals that represent integers. The block represents each integer using a group of $K = \log_2(M)$ bits, where K represents the number of bits per symbol. The output vector length must be an integer multiple of K.

In binary output mode, the symbols can be either binary-demapped or Gray-demapped. The **Constellation ordering** parameter indicates how the block maps an integer to a corresponding group of K output bits. See the reference pages for the M-DPSK Modulator Baseband and M-PSK Modulator Baseband blocks for details.

Dialog Box



M-ary number

The number of possible modulated symbols that can immediately follow a given symbol.

Output type

Determines whether the output consists of integers or groups of bits.

Constellation ordering

Determines how the block maps each integer to a group of output bits.

Phase rotation (rad)
This phase difference between the current and previous modulated symbols that results in an output of zero.

Output data type
When the parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is of type `single` or `double`.

For integer outputs, this block can output the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`. For bit outputs, output can be `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, or `double`.

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 • Boolean when **Output type** set to `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

M-DPSK Modulator Baseband

See Also

DBPSK Demodulator Baseband, DQPSK Demodulator Baseband, M-PSK Demodulator Baseband

References

[1] Pawula, R. F., "On M-ary DPSK Transmission Over Terrestrial and Satellite Channels," *IEEE Transactions on Communications*, Vol. COM-32, July 1984, 752-761.

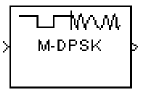
Purpose

Modulate using M-ary differential phase shift keying method

Library

PM, in Digital Baseband sublibrary of Modulation

Description



The M-DPSK Modulator Baseband block modulates using the M-ary differential phase shift keying method. The output is a baseband representation of the modulated signal. The **M-ary number** parameter, M, is the number of possible output symbols that can immediately follow a given output symbol.

The input must be a discrete-time signal. For integer inputs, the block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`. For bit inputs, the block can accept `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, and `double`.

The input can be either bits or integers, which are binary-mapped or Gray-mapped into symbols.

This block accepts column vector input signals. For a bit input, the input width must be an integer multiple of the number of bits per symbol.

Integer-Valued Signals and Binary-Valued Signals

If you set the **Input type** parameter to `Integer`, then valid input values are integers between 0 and M-1. In this case, the input can be either a scalar or a frame-based column vector. If the first input is k_1, then the modulated symbol is

$$\exp\left(j\theta + j2\pi\frac{k_1}{m}\right)$$

where θ represents the **Phase rotation** parameter. If a successive input is *k*, then the modulated symbol is

$$\exp\left(j\theta + j2\pi\frac{k}{m}\right)\cdot(\text{previous modulated symbol})$$

When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

where

K represents the number of bits per symbol.

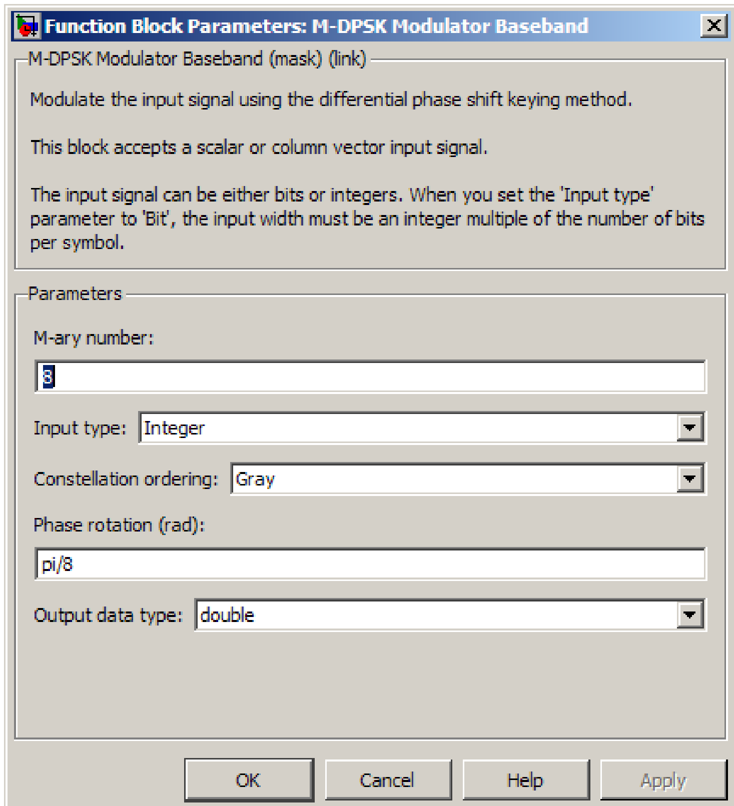
The input vector length must be an integer multiple of K. In this configuration, the block accepts a group of K bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol for each group of K bits.

The input can be a column vector with a length that is an integer multiple of K.

In binary input mode, the **Constellation ordering** parameter indicates how the block maps a group of K input bits to a corresponding phase difference. The `Binary` option uses a natural binary-to-integer mapping, while the `Gray` option uses a Gray-coded assignment of phase differences. For example, the following table indicates the assignment of phase difference to three-bit inputs, for both the `Binary` and `Gray` options. θ is the **Phase rotation** parameter. The phase difference is between the previous symbol and the current symbol.

Current Input	Binary-Coded Phase Difference	Gray-Coded Phase Difference
[0 0 0]	jθ	jθ
[0 0 1]	jθ + jπ/4	jθ + jπ/4
[0 1 0]	jθ + jπ2/4	jθ + jπ3/4
[0 1 1]	jθ + jπ3/4	jθ + jπ2/4
[1 0 0]	jθ + jπ4/4	jθ + jπ7/4
[1 0 1]	jθ + jπ5/4	jθ + jπ6/4
[1 1 0]	jθ + jπ6/4	jθ + jπ4/4
[1 1 1]	jθ + jπ7/4	jθ + jπ5/4

For more details about the `Binary` and `Gray` options, see the reference page for the M-PSK Modulator Baseband block. The signal constellation for that block corresponds to the arrangement of phase differences for this block.



Dialog Box

M-ary number

The number of possible output symbols that can immediately follow a given output symbol.

Input type
Indicates whether the input consists of integers or groups of bits. If this parameter is set to `Bit`, then the **M-ary number** parameter must be 2^K for some positive integer K.

Constellation ordering
Determines how the block maps each group of input bits to a corresponding integer.

Phase rotation (rad)
The phase difference between the previous and current modulated symbols when the input is zero.

Output data type
The output data type can be either `single` or `double`. By default, the block sets this to `double`.

Supported Data Types

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

Pair Block

M-DPSK Demodulator Baseband

See Also

DBPSK Modulator Baseband, DQPSK Modulator Baseband, M-PSK Modulator Baseband

References

[1] Pawula, R. F., "On M-ary DPSK Transmission Over Terrestrial and Satellite Channels," *IEEE Transactions on Communications*, Vol. COM-32, July 1984, 752-761.

Memoryless Nonlinearity

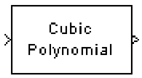
Purpose

Apply memoryless nonlinearity to complex baseband signal

Library

RF Impairments

Description



The Memoryless Nonlinearity block applies a memoryless nonlinearity to a complex, baseband signal. You can use the block to model radio frequency (RF) impairments to a signal at the receiver.

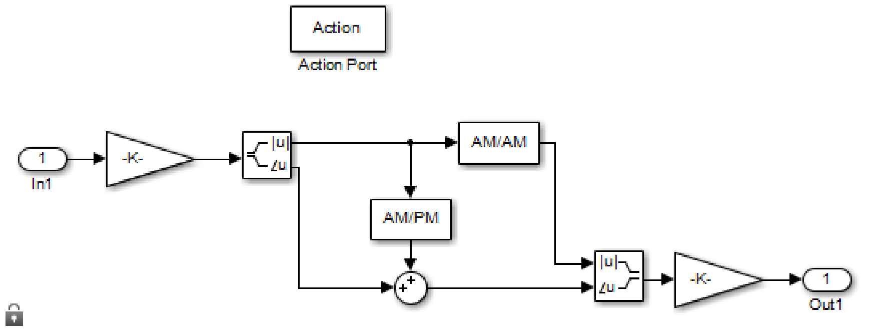
This block accepts a column vector input signal.

Note All values of power assume a nominal impedance of 1 ohm.

The Memoryless Nonlinearity block provides five different methods for modeling the nonlinearity, which you specify by the **Method** parameter. The options for the **Method** parameter are

- `Cubic polynomial`
- `Hyperbolic tangent`
- `Saleh model`
- `Ghorbani model`
- `Rapp model`

The bock implements these five methods using subsystems underneath the block mask. For each of the first four methods, the nonlinearity subsystem has the same basic structure, as shown in the following figure.

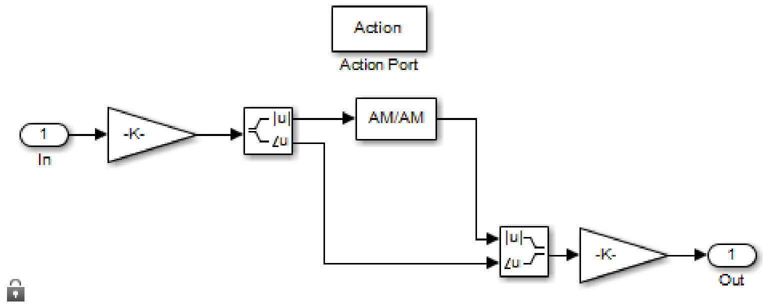


Nonlinearity Subsystem

For the first four methods, each subsystem applies a nonlinearity to the input signal as follows:

1. Multiply the signal by a gain factor.
2. Split the complex signal into its magnitude and angle components.
3. Apply an AM/AM conversion to the magnitude of the signal, according to the selected **Method**, to produce the magnitude of the output signal.
4. Apply an AM/PM conversion to the phase of the signal, according to the selected **Method**, and adds the result to the angle of the signal to produce the angle of the output signal.
5. Combine the new magnitude and angle components into a complex signal and multiply the result by a gain factor, which is controlled by the **Linear gain** parameter.

Each subsystem implements the AM/AM and AM/PM conversions differently, according to the Method you specify. The Rapp model does not apply a phase change to the input signal. The nonlinearity subsystem for Rapp model has following structure:



Nonlinearity Subsystem for Rapp Model

The Rapp Subsystem applies nonlinearity as follows:

1. Multiply the signal by a gain factor.
2. Split the complex signal into its magnitude and angle components.
3. Apply an AM/AM conversion to the magnitude of the signal, according to the selected **Method**, to produce the magnitude of the output signal.
4. Combine the new magnitude and angle components into a complex signal and multiply the result by a gain factor, which is controlled by the **Linear gain** parameter.

If you want to see exactly how the Memoryless Nonlinearity block implements the conversions for a specific method, you can view the AM/AM and AM/PM subsystems that implement these conversions as follows:

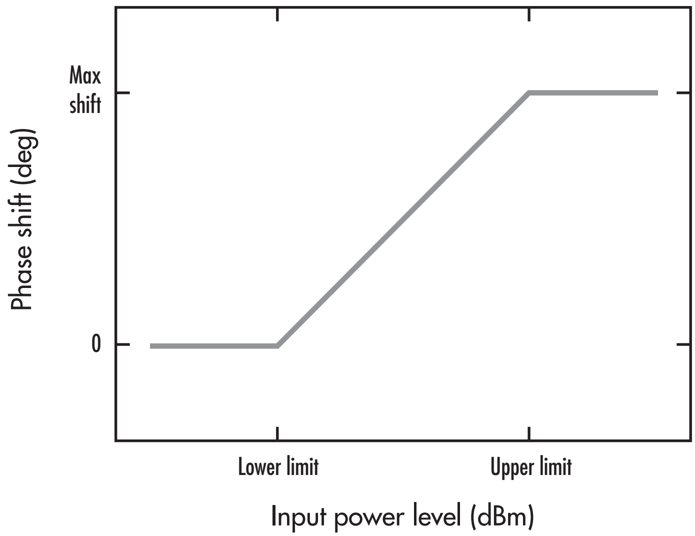
1. Right-click on the Memoryless Nonlinearity block and select **Mask** > **Look under mask**. This displays the block's configuration underneath the mask. The block contains five subsystems corresponding to the five nonlinearity methods.

2 Double-click the subsystem for the method you are interested in. This displays the subsystem shown in the preceding figure, Nonlinearity Subsystem on page 2-513.

3 Double-click on one of the subsystems labeled AM/AM or AM/PM to view how the block implements the conversions.

AM/PM Characteristics of the Cubic Polynomial and Hyperbolic Tangent Methods

The following illustration shows the AM/PM behavior for the `Cubic polynomial` and `Hyperbolic tangent` methods:



The AM/PM conversion scales linearly with input power value between the lower and upper limits of the input power level (specified by **Lower input power limit for AM/PM conversion (dBm)** and **Upper input power limit for AM/PM conversion (dBm)**). Beyond these values, AM/PM conversion is constant at the values corresponding to the lower and upper input power limits, which are zero and

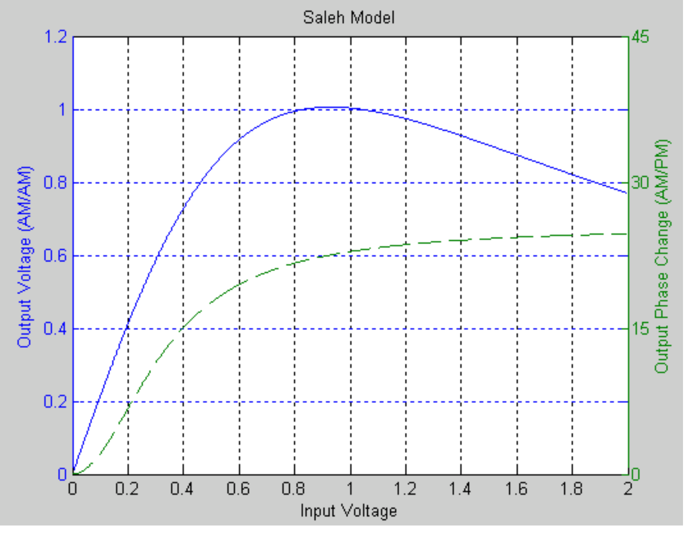
$$(\text{AM/PM conversion}) \cdot (\text{upper input power limit} - \text{lower input power limit}),$$

respectively.

AM/AM and AM/PM Characteristics of the Saleh Method

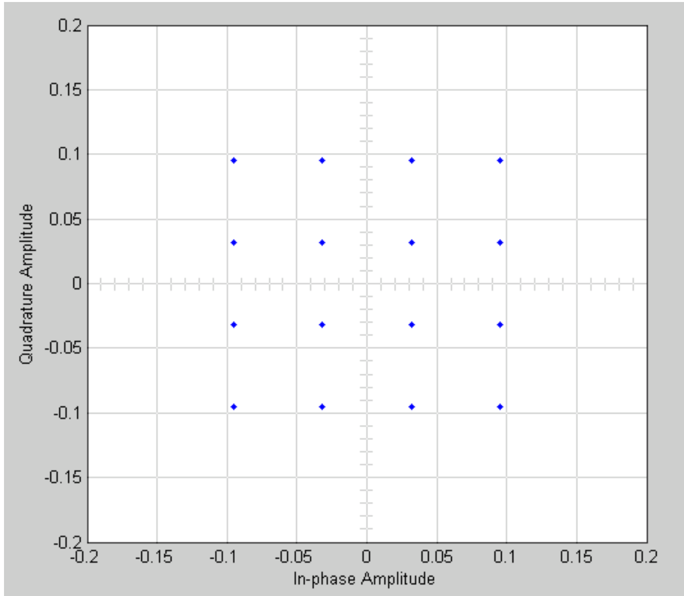
The following figure shows, for the Saleh method, plots of

- Output voltage against input voltage for the AM/AM conversion
- Output phase against input voltage for the AM/PM conversion

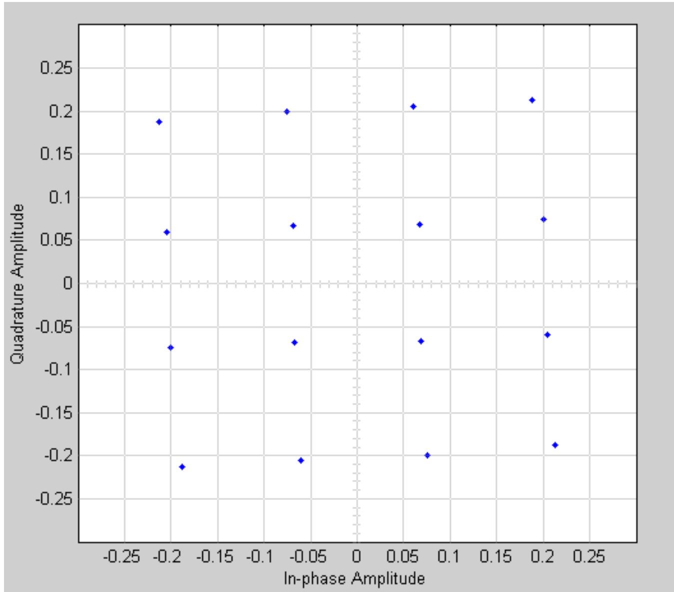


Example with 16-ary QAM

You can see the effect of the Memoryless Nonlinearity block on a signal modulated by 16-ary quadrature amplitude modulation (QAM) in a scatter plot. The constellation for 16-ary QAM without the effect of the Memoryless Nonlinearity block is shown in the following figure:



You can generate a scatter plot of the same signal after it passes through the Memoryless Nonlinearity block, with the **Method** parameter set to `Saleh Model`, as shown in the following figure.



This plot is generated by the model described in “Illustrate RF Impairments That Distort a Signal” with the following parameter settings for the Rectangular QAM Modulator Baseband block:

- **Normalization method** set to `Average Power`
- **Average power (watts)** set to `1e-2`

The following sections discuss parameters specific to the Saleh, Ghorbani, and Rapp models.

Parameters for the Saleh Model

The **Input scaling (dB)** parameter scales the input signal before the nonlinearity is applied. The block multiplies the input signal by the parameter value, converted from decibels to linear units. If you set the parameter to be the inverse of the input signal amplitude, the scaled signal has amplitude normalized to 1.

The AM/AM parameters, alpha and beta, are used to compute the amplitude gain for an input signal using the following function:

$$F_{AM/AM}(u) = \frac{\text{alpha} * u}{1 + \text{beta} * u^2}$$

where u is the magnitude of the scaled signal.

The AM/PM parameters, alpha and beta, are used to compute the phase change for an input signal using the following function:

$$F_{AM/PM}(u) = \frac{\text{alpha} * u^2}{1 + \text{beta} * u^2}$$

where u is the magnitude of the scaled signal. Note that the AM/AM and AM/PM parameters, although similarly named alpha and beta, are distinct.

The **Output scaling (dB)** parameter scales the output signal similarly.

Parameters for the Ghorbani Model

The **Input scaling (dB)** parameter scales the input signal before the nonlinearity is applied. The block multiplies the input signal by the parameter value, converted from decibels to linear units. If you set the parameter to be the inverse of the input signal amplitude, the scaled signal has amplitude normalized to 1.

The AM/AM parameters, $[x_1\ x_2\ x_3\ x_4]$, are used to compute the amplitude gain for an input signal using the following function:

$$F_{AM/AM}(u) = \frac{x_1 u^{x_2}}{1 + x_3 u^{x_2}} + x_4 u$$

where u is the magnitude of the scaled signal.

The AM/PM parameters, $[y_1\ y_2\ y_3\ y_4]$, are used to compute the phase change for an input signal using the following function:

$$F_{AM/PM}(u) = \frac{y_1 u^{y_2}}{1 + y_3 u^{y_2}} + y_4 u$$

where u is the magnitude of the scaled signal.

The **Output scaling (dB)** parameter scales the output signal similarly.

Parameters for the Rapp Model

The **Linear gain (dB)** parameter scales the input signal before the nonlinearity is applied. The block multiplies the input signal by the parameter value, converted from decibels to linear units. If you set the parameter to be the inverse of the input signal amplitude, the scaled signal has amplitude normalized to 1.

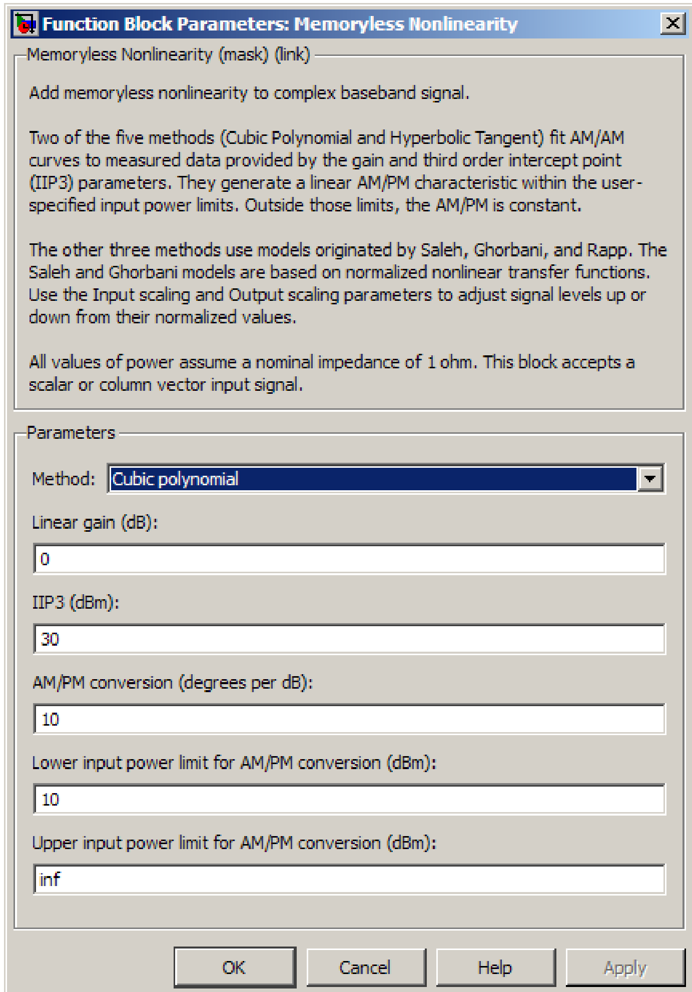
The **Smoothness factor** and **Output saturation level** parameters are used to compute the amplitude gain for the input signal:

$$F_{AM/AM}(u) = \frac{u}{\left(1 + \left(\frac{u}{O_{sat}}\right)^{2S}\right)^{1/2S}}$$

where u is the magnitude of the scaled signal, S is the **Smoothness factor**, and O_{sat} is the **Output saturation level**.

The Rapp model does not apply a phase change to the input signal.

The **Output saturation level** parameter limits the output signal level.

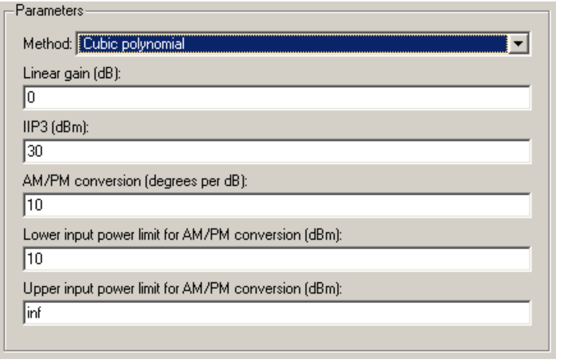


Dialog Box

Method

The nonlinearity method.

The following describes specific parameters for each method.



Linear gain (db)
Scalar specifying the linear gain for the output function.

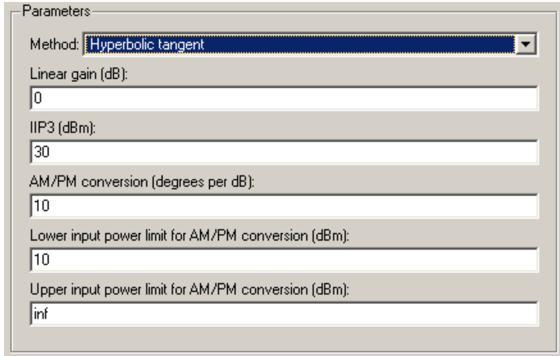
IIP3 (dBm)
Scalar specifying the third order intercept.

AM/PM conversion (degrees per dB)
Scaler specifying the AM/PM conversion in degrees per decibel.

Lower input power limit (dBm)
Scalar specifying the minimum input power for which AM/PM conversion scales linearly with input power value. Below this value, the phase shift resulting from AM/PM conversion is zero.

Upper input power limit (dBm)
Scalar specifying the maximum input power for which AM/PM conversion scales linearly with input power value. Above this value, the phase shift resulting from AM/PM conversion is constant. The value of this maximum shift is given by:

$$(\text{AM/PM conversion}) \cdot (\text{upper input power limit} - \text{lower input power limit})$$



Linear gain (db)
: Scalar specifying the linear gain for the output function.

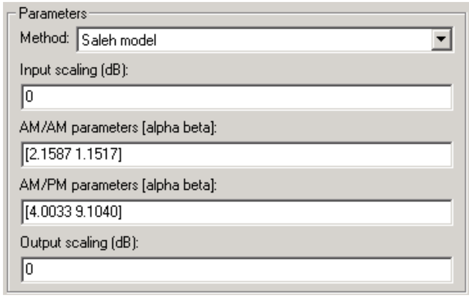
IIP3 (dBm)
: Scalar specifying the third order intercept.

AM/PM conversion (degrees per dB)
: Scalar specifying the AM/PM conversion in degrees per decibel.

Lower input power limit (dBm)
: Scalar specifying the minimum input power for which AM/PM conversion scales linearly with input power value. Below this value, the phase shift resulting from AM/PM conversion is zero.

Upper input power limit (dBm)
: Scalar specifying the maximum input power for which AM/PM conversion scales linearly with input power value. Above this value, the phase shift resulting from AM/PM conversion is constant. The value of this maximum shift is given by:

$$(\text{AM/PM conversion}) \cdot (\text{upper input power limit} - \text{lower input power limit})$$

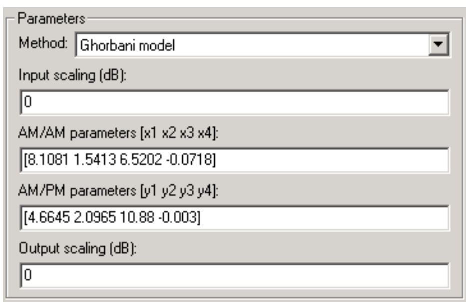


Input scaling (dB)
Number that scales the input signal level.

AM/AM parameters [alpha beta]
Vector specifying the AM/AM parameters.

AM/PM parameters [alpha beta]
Vector specifying the AM/PM parameters.

Output scaling (dB)
Number that scales the output signal level.

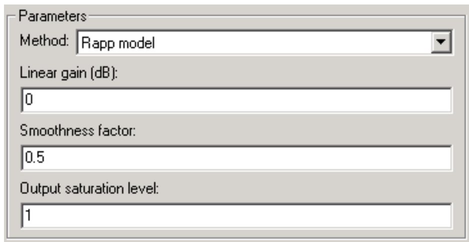


Input scaling (dB)
Number that scales the input signal level.

AM/AM parameters [x1 x2 x3 x4]
Vector specifying the AM/AM parameters.

AM/PM parameters [y1 y2 y3 y4]
Vector specifying the AM/PM parameters.

Output scaling (dB)
Number that scales the output signal level.



Linear gain (db)
Scalar specifying the linear gain for the output function.

Smoothness factor
Scalar specifying the smoothness factor

Output saturation level
Scalar specifying the the output saturation level.

See Also

I/Q Imbalance

Reference

[1] Saleh, A.A.M., "Frequency-independent and frequency-dependent nonlinear models of TWT amplifiers," IEEE Trans. Communications, vol. COM-29, pp.1715-1720, November 1981.

[2] A. Ghorbani, and M. Sheikhan, "The effect of Solid State Power Amplifiers (SSPAs) Nonlinearities on MPSK and M-QAM Signal Transmission", Sixth Int'l Conference on Digital Processing of Signals in Comm., 1991, pp. 193-197.

[3] C. Rapp, "Effects of HPA-Nonlinearity on a 4-DPSK/OFDM-Signal for a Digitial Sound Broadcasting System", in Proceedings of the Second

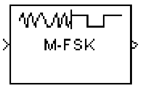
European Conference on Satellite Communications, Liege, Belgium, Oct. 22-24, 1991, pp. 179-184.

M-FSK Demodulator Baseband

Purpose Demodulate FSK-modulated data

Library FM, in Digital Baseband sublibrary of Modulation

Description



The M-FSK Demodulator Baseband block demodulates a signal that was modulated using the M-ary frequency shift keying method. The input is a baseband representation of the modulated signal. The input and output for this block are discrete-time signals. This block accepts a scalar value or column vector input signal of type `single` or `double`. For information about the data types each block port supports, see "Supported Data Types" on page 2-533.

The **M-ary number** parameter, M, is the number of frequencies in the modulated signal. The **Frequency separation** parameter is the distance, in Hz, between successive frequencies of the modulated signal.

The M-FSK Demodulator Baseband block implements a non-coherent energy detector. To obtain the same BER performance as that of coherent FSK demodulation, use the CPFSK Demodulator Baseband block.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Output type** parameter to `Integer`, the block outputs integer values between `0` and M-`1`. M represents the **M-ary number** block parameter.

When you set the **Output type** parameter to `Bit`, the block outputs binary-valued signals that represent integers. The block represents each integer using a group of $K = \log_2(M)$ bits, where K represents the number of bits per symbol. The output vector length must be an integer multiple of K.

The **Symbol set ordering** parameter indicates how the block maps a symbol to a group of K output bits. When you set the parameter to `Binary`, the block maps the integer, I, to [u(1) u(2) ... u(K)] bits, where the individual u(i) are given by

$$I = \sum_{i=1}^{K} u(i) 2^{K-i}$$

$u(1)$ is the most significant bit.

For example, if $M = 8$, you set **Symbol set ordering** to `Binary`, and the demodulated integer symbol value is 6, then the binary output word is [1 1 0].

When you set **Symbol set ordering** to `Gray`, the block assigns binary outputs from points of a predefined Gray-coded signal constellation. The predefined M-ary Gray-coded signal constellation assigns the binary representation

```
M = 8; P = [0:M-1]';
de2bi(bitxor(P,floor(P/2)), log2(M),'left-msb')
```

to the P^{th} integer.

The typical Binary to Gray mapping for $M = 8$ is shown in the following tables.

Binary to Gray Mapping for Bits

Binary Code	Gray Code
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Binary to Gray Mapping for Integers

Binary Code	Gray Code
0	0
1	1
2	3
3	2
4	6
5	7
6	5
7	4

Whether the output is an integer or a binary representation of an integer, the block maps the highest frequency to the integer 0 and maps the lowest frequency to the integer M-1. In baseband simulation, the lowest frequency is the negative frequency with the largest absolute value.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input width must be an integer multiple of the **Samples per symbol** parameter value, and the input can be a column vector.

- When you set **Output type** to `Bit`, the output width is K times the number of input symbols.
- When you set **Output type** to `Integer`, the output width is the number of input symbols.

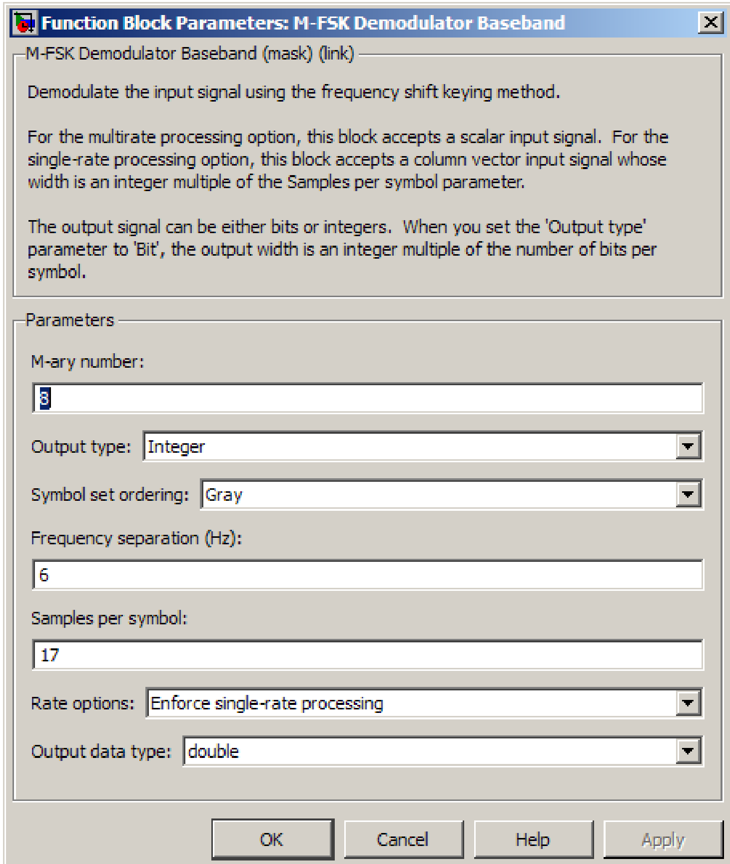
Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is the product of the input sample time and the **Samples per symbol** parameter value.

- When you set **Output type** to `Bit`, the output width equals the number of bits per symbol.
- When you set **Output type** to `Integer`, the output is a scalar.

To run the M-FSK Demodulator block in multirate mode, set **Tasking mode for periodic sample times** (in **Simulation > Configuration Parameters > Solver**) to `SingleTasking`.

Dialog Box



M-ary number

The number of frequencies in the modulated signal.

Output type

Determines whether the output consists of integers or groups of bits. If this parameter is set to `Bit`, then the **M-ary number** parameter must be 2^K for some positive integer K.

Symbol set ordering
: Determines how the block maps each integer to a group of output bits.

Frequency separation (Hz)
: The distance between successive frequencies in the modulated signal.

Samples per symbol
: The number of input samples that represent each modulated symbol.

Rate options
: Select the rate processing method for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample times. The block implements the rate change by making a size change at the output when compared to the input. The output width is the number of symbols (which is given by dividing the input length by the **Samples per symbol** parameter value when the **Output type** parameter is set to `Integer`).
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output period is the same as the symbol period and equals the product of the input period and the **Samples per symbol** parameter value.

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 *Communications System Toolbox Release Notes* for more information.

For more information, see Single-Rate Processing and Multirate Processing in the Description section of this page.

Output data type

The output type of the block can be specified here as `boolean`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `double`. By default, the block sets this to `double`.

Supported Data Types

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

Pair Block

M-FSK Modulator Baseband

See Also

CPFSK Demodulator Baseband

M-FSK Modulator Baseband

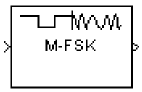
Purpose

Modulate using M-ary frequency shift keying method

Library

FM, in Digital Baseband sublibrary of Modulation

Description



The M-FSK Modulator Baseband block modulates using the M-ary frequency shift keying method. The output is a baseband representation of the modulated signal. For information about the data types each block port supports, see "Supported Data Types" on page 2-540.

To prevent aliasing from occurring in the output signal, set the sampling frequency greater than the product of *M* and the **Frequency separation** parameter. Sampling frequency is **Samples per symbol** divided by the input symbol period (in seconds).

Integer-Valued Signals and Binary-Valued Signals

The input and output signals for this block are discrete-time signals.

When you set the **Input type** parameter to `Integer`, the block accepts integer values between `0` and *M*-`1`. *M* represents the **M-ary number** block parameter.

When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

where

K represents the number of bits per symbol.

The input vector length must be an integer multiple of *K*. In this configuration, the block accepts a group of *K* bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol, oversampled by the **Samples per symbol** parameter value, for each group of *K* bits.

The **Symbol set ordering** parameter indicates how the block maps a group of *K* input bits to a corresponding symbol. When you set the parameter to `Binary`, the block maps [$u(1)$ $u(2)$ ... $u(\mathrm{K})$] to the integer

$$\sum_{i=1}^{K} u(i) 2^{K-i}$$

and assumes that this integer is the input value. $u(1)$ is the most significant bit.

If you set $M = 8$, **Symbol set ordering** to `Binary`, and the binary input word is [1 1 0], the block converts [1 1 0] to the integer 6. The block produces the same output when the input is 6 and the **Input type** parameter is `Integer`.

When you set **Symbol set ordering** to `Gray`, the block uses a Gray-coded arrangement and assigns binary inputs to points of a predefined Gray-coded signal constellation. The predefined M-ary Gray-coded signal constellation assigns the binary representation

```
M = 8; P = [0:M-1]';
de2bi(bitxor(P,floor(P/2)), log2(M),'left-msb')
```

to the P^{th} integer.

The following tables show the typical Binary to Gray mapping for $M = 8$.

Binary to Gray Mapping for Bits

Binary Code	Gray Code
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Binary to Gray Mapping for Integers

Binary Code	Gray Code
0	0
1	1
2	3
3	2
4	6
5	7
6	5
7	4

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. In this mode, the input to the block can be multiple symbols.

- When you set **Input type** to `Integer`, the input can be a column vector, the length of which is the number of input symbols.
- When you set **Input type** to `Bit`, the input width must be an integer multiple of K, the number of bits per symbol.

The output width equals the product of the number of input symbols and the **Samples per symbol** parameter value.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

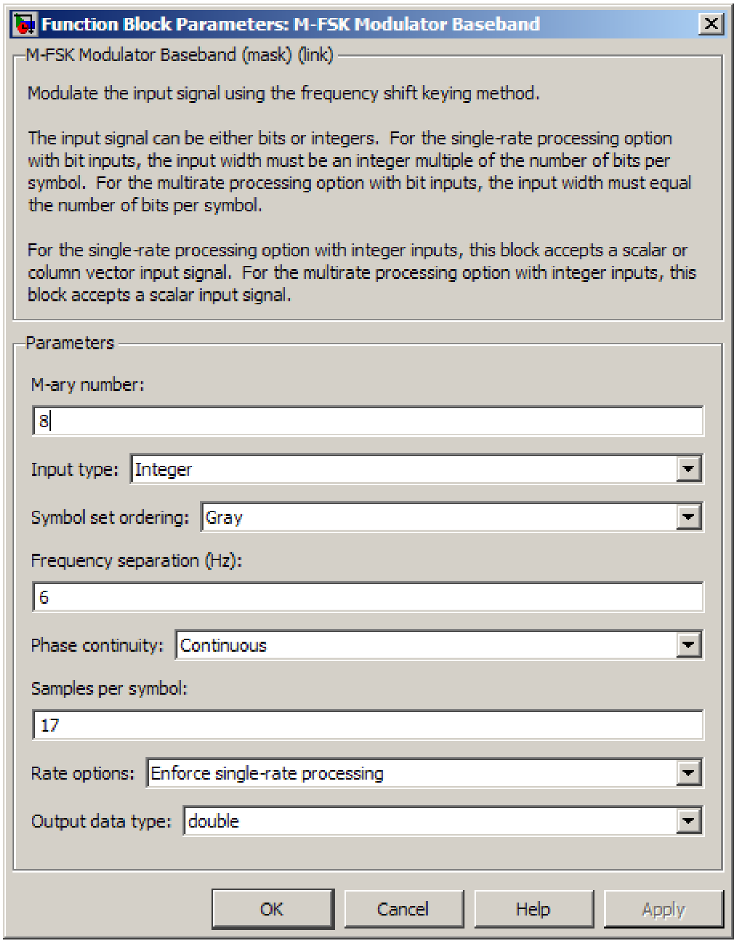
- When you set **Input type** to `Integer`, the input must be a scalar.

- When you set **Input type** to `Bit`, the input width must equal the number of bits per symbol.

The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

To run the M-FSK Modulator block in multirate mode, set **Tasking mode for periodic sample times** (in **Simulation > Configuration Parameters > Solver**) to `SingleTasking`.

Dialog Box



M-ary number

The number of frequencies in the modulated signal.

Input type

Indicates whether the input consists of integers or groups of bits. If you set this parameter to `Bit`, then the **M-ary number** parameter must be 2^K for some positive integer K.

Symbol set ordering
Determines how the block maps each group of input bits to a corresponding integer.

Frequency separation (Hz)
The distance between successive frequencies in the modulated signal.

Phase continuity
Determines whether the modulated signal changes phases in a continuous or discontinuous way.

If you set the **Phase continuity** parameter to `Continuous`, then the modulated signal maintains its phase even when it changes its frequency. If you set the **Phase continuity** parameter to `Discontinuous`, then the modulated signal comprises portions of M sinusoids of different frequencies. Thus, a change in the input value sometimes causes a change in the phase of the modulated signal.

Samples per symbol
The number of output samples that the block produces for each integer or binary word in the input.

Rate options
Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

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 *Communications System Toolbox Release Notes* for more information.

Output data type

You can specify the output type of the block as either a `double` or a `single`. By default, the block sets this value to `double`.

Supported Data Types

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

Pair Block

M-FSK Demodulator Baseband

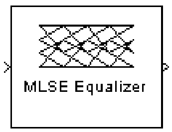
See Also

CPFSK Modulator Baseband

Purpose
Equalize using Viterbi algorithm

Library
Equalizer Block

Description



The MLSE Equalizer block uses the Viterbi algorithm to equalize a linearly modulated signal through a dispersive channel. The block processes input frames and outputs the maximum likelihood sequence estimate (MLSE) of the signal, using an estimate of the channel modeled as a finite input response (FIR) filter.

This block supports `single` and `double` data types.

Channel Estimates

The channel estimate takes the form of a column vector containing the coefficients of an FIR filter in descending order of powers. The length of this vector is the channel memory, which must be a multiple of the block's **Samples per input symbol** parameter.

To specify the channel estimate vector, use one of these methods:

- Set **Specify channel via** to `Dialog` and enter the vector in the **Channel coefficients** field.
- Set **Specify channel via** to `Input port` and the block displays an additional input port, labeled `Ch`, which accepts a column vector input signal.

Signal Constellation

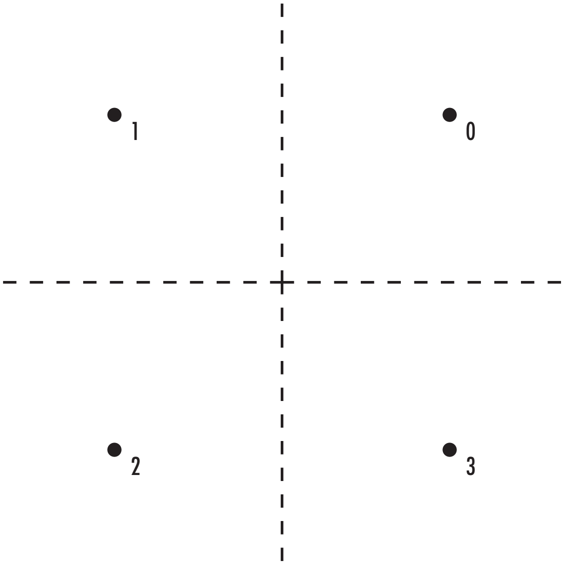
The **Signal constellation** parameter specifies the constellation for the modulated signal, as determined by the modulator in your model. **Signal constellation** is a vector of complex numbers, where the kth complex number in the vector is the constellation point to which the modulator maps the integer k-1.

Note The sequence of constellation points must be consistent between the modulator in your model and the **Signal constellation** parameter in this block.

For example, to specify the constellation given by the mapping

$$0 \rightarrow +1+i$$
$$1 \rightarrow -1+i$$
$$2 \rightarrow -1-i$$
$$3 \rightarrow +1-i$$

set **Constellation points** to `[1+i, -1+i, -1-i, 1-i]`. Note that the sequence of numbers in the vector indicates how the modulator maps integers to the set of constellation points. The labeled constellation is shown below.



Preamble and Postamble

If your data is accompanied by a preamble (prefix) or postamble (suffix), then configure the block accordingly:

- If you select **Input contains preamble**, then the **Expected preamble** parameter specifies the preamble that you expect to precede the data in the input signal.
- If you check the **Input contains postamble**, then the **Expected postamble** parameter specifies the postamble that you expect to follow the data in the input signal.

The **Expected preamble** or **Expected postamble** parameter must be a vector of integers between 0 and M-1, where M is the number of constellation points. An integer value of k-1 in the vector corresponds to the kth entry in the **Constellation points** vector and, consequently, to a modulator input of k-1.

The preamble or postamble must already be included at the beginning or end, respectively, of the input signal to this block. If necessary, you can concatenate vectors in Simulink software using the Matrix Concatenation block.

To learn how the block uses the preamble and postamble, see “"Reset Every Frame" Operation Mode” on page 2-543 below.

"Reset Every Frame" Operation Mode

One way that the Viterbi algorithm can transition between successive frames is called `Reset every frame` mode. You can choose this mode using the **Operation mode** parameter.

In `Reset every frame` mode, the block decodes each frame of data independently, resetting the state metric at the end of each frame. The traceback decoding always starts at the state with the minimum state metric.

The initialization of state metrics depends on whether you specify a preamble and/or postamble:

- If you do not specify a preamble, the decoder initializes the metrics of all states to 0 at the beginning of each frame of data.
- If you specify a preamble, the block uses it to initialize the state metrics at the beginning of each frame of data. More specifically, the block decodes the preamble and assigns a metric of 0 to the decoded state. If the preamble does not decode to a unique state -- that is, if the length of the preamble is less than the channel memory -- the decoder assigns a metric of 0 to all states that can be represented by the preamble. Whenever you specify a preamble, the traceback path ends at one of the states represented by the preamble.
- If you do not specify a postamble, the traceback path starts at the state with the smallest metric.
- If you specify a postamble, the traceback path begins at the state represented by the postamble. If the postamble does not decode to a unique state, the decoder identifies the smallest of all possible decoded states that are represented by the postamble and begins traceback decoding at that state.

Note In `Reset every frame` mode, the input to the MLSE Equalizer block must contain at least T symbols, not including an optional preamble, where T is the **Traceback depth** parameter.

Continuous Operation Mode

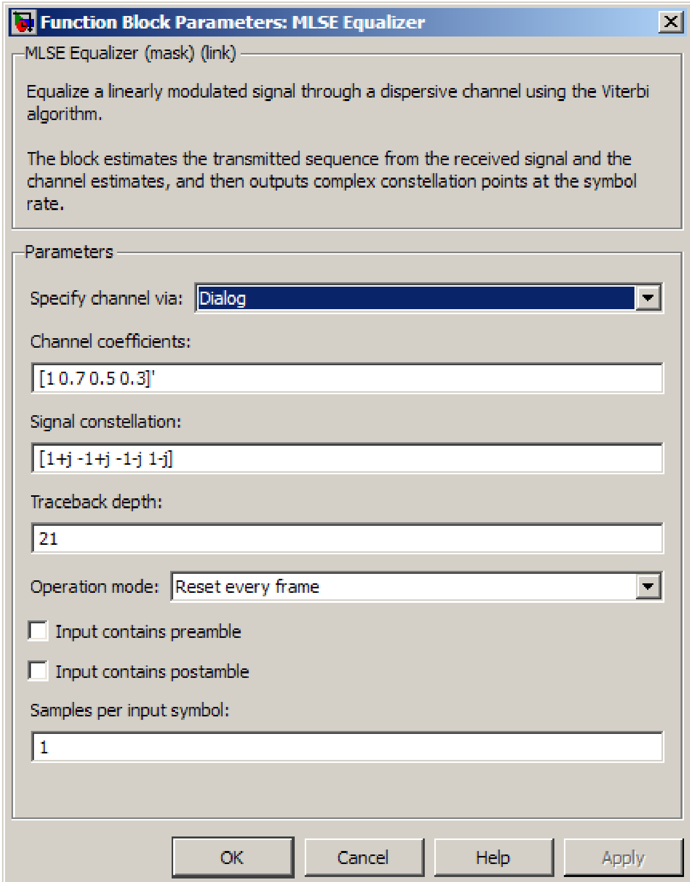
An alternative way that the Viterbi algorithm can transition between successive frames is called `Continuous with reset option` mode. You can choose this mode using the **Operation mode** parameter.

In `Continuous with reset option` mode, the block initializes the metrics of all states to 0 at the beginning of the simulation. At the end of each frame, the block saves the internal state metric for use in computing the traceback paths in the next frame.

If you select **Enable the reset input port**, the block displays another input port, labeled `Rst`. In this case, the block resets the state metrics whenever the scalar value at the `Rst` port is nonzero.

Decoding Delay

The MLSE Equalizer block introduces an output delay equal to the **Traceback depth** in the `Continuous with reset option` mode, and no delay in the `Reset every frame` mode.



Dialog Box

Specify channel via

The method for specifying the channel estimate. If you select `Input port`, the block displays a second input port that receives the channel estimate. If you select `Dialog`, you can specify the channel estimate as a vector of coefficients for an FIR filter in the **Channel coefficients** field.

Channel coefficients
Vector containing the coefficients of the FIR filter that the block uses for the channel estimate. This field is visible only if you set **Specify channel via** to `Dialog`.

Signal constellation
Vector of complex numbers that specifies the constellation for the modulation.

Traceback depth
The number of trellis branches (equivalently, the number of symbols) the block uses in the Viterbi algorithm to construct each traceback path.

Operation mode
The operation mode of the Viterbi decoder. Choices are `Continuous with reset option` and `Reset every frame`.

Input contains preamble
When checked, you can set the preamble in the **Expected preamble** field. This option appears only if you set **Operation mode** to `Reset every frame`.

Expected preamble
Vector of integers between 0 and M-1 representing the preamble, where M is the size of the constellation. This field is visible and active only if you set **Operation mode** to `Reset every frame` and then select **Input contains preamble**.

Input contains postamble
When checked, you can set the postamble in the **Expected postamble** field. This option appears only if you set **Operation mode** to `Reset every frame`.

Expected postamble
Vector of integers between 0 and M-1 representing the postamble, where M is the size of the constellation. This field is visible and active only if you set **Operation mode** to `Reset every frame` and then select **Input contains postamble**.

Samples per input symbol
The number of input samples for each constellation point.

Enable the reset input port
When you check this box, the block has a second input port labeled `Rst`. Providing a nonzero input value to this port causes the block to set its internal memory to the initial state before processing the input data. This option appears only if you set **Operation mode** to `Continuous with reset option`.

Example

MLSE Equalization with dynamically changing channel

This example shows how to equalize the effects of a Multipath Rayleigh Fading Channel block. Maximum Likelihood Sequence Estimation (MLSE) estimates the data the model transmits through a time varying dispersive channel with the least possible number of errors. This model inputs the dynamically evolving channel coefficients of a two-path channel to the MLSE Equalizer block. The model shows the MLSE block being used in a typical multipath wireless Rayleigh channel. It applies the same channel estimate to 50 samples in the frame that is processed by the MLSE Equalizer. This is similar to a practical system, where the training sequence is transmitted in regular intervals and a channel estimate is used until the next training symbol is transmitted.

To open the example, type `doc_mlse_dynamic_coeffs` at the MATLAB command line.

Block Parameters in the model

- The sample time of the Bernoulli Binary Generator block is set to 5e-6, which corresponds to a bit rate of 200 kbps, and a QPSK symbol rate of 100 ksym/sec.

- The Multipath Rayleigh Fading Channel block has a **Maximum Doppler shift** of 30 Hz, which is a realistic physical value. The Delay vector of the MRFC block is [0 1e-5], which corresponds to two consecutive sample times of the input QPSK symbol data. This reflects the simplest delay vector for a two-path channel. The **Average path gain vector** is set arbitrarily to [0 -10]. The gain vector is normalized to 0 dB, so that the average power input to the AWGN block is 1 W.

- The MLSE Equalizer block has the **Traceback depth** set to 10 and may be varied to study its effect on Bit Error rate (BER).
- The QPSK Demodulator accepts an N-by-1 input frame and generates a 2N-by-1 output frame. This, along with the traceback depth of 10 results in a delay of 20 bits. The model performs frame-based processing with 100 samples per frame. Thus, there is a delay of 100 bits inherent in the model. The combined receive delay of 120 is set in the **Receive delay** parameter of the Error Rate Calculation block, aligning the samples.

Block Parameters in the model

The sample time of the Bernoulli Binary Generator block is set to 5e-6, which corresponds to a bit rate of 200 kbps, and a QPSK symbol rate of 100 ksym/sec. Multipath Rayleigh Fading Channel (MRFC) block: The MRFC block has a max Doppler shift of 30 Hz, which is a realistic physical value. The Delay vector of the MRFC block is [0 1e-5], which corresponds to two consecutive sample times of the input QPSK symbol data. This reflects the simplest delay vector for a two-path channel. The Gain vector of the MRFC block is set arbitrarily to [0 -10]. The gain vector is normalized to 0 dB, so that the average power input to the AWGN block is 1 W.

See Also

LMS Linear Equalizer, LMS Decision Feedback Equalizer, RLS Linear Equalizer, RLS Decision Feedback Equalizer, CMA Equalizer

References

[1] Proakis, John G., *Digital Communications*, Fourth edition, New York, McGraw-Hill, 2001.

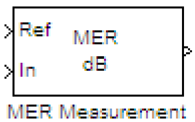
[2] Steele, Raymond, Ed., *Mobile Radio Communications*, Chichester, England, Wiley, 1996.

MER Measurement

Purpose Measure signal-to-noise ratio (SNR) in digital modulation applications

Library Utility Blocks

Description



The Modulation Error Ratio (MER) is a measure of the signal-to-noise ratio (SNR) in digital modulation applications. You can use these types of measurements to determine system performance in communications applications. For example, determining if an EDGE system conforms to 3GPP radio transmission standards requires accurate MER, Minimum MER, and 95th percentile for the MER measurements. The block measures all outputs in decibels (dB).

The MER block receives an ideal input signal (at reference port, `Ref`) and an AWGN corrupted signal (at input port, `In`). The MER block then outputs a measure of the modulation accuracy by comparing these inputs. The Modulation Error Ratio is the ratio of the average reference signal power to the mean square error. This ratio corresponds to the SNR of the AWGN channel.

The block output defaults to MER in decibels (dB), with an option of `Output minimum MER` or `Output X-percentile MER` values. The minimum MER represents the best-case MER value per burst. For the X-percentile option, you can select to output the number of symbols processed in the percentile computations.

The following table shows the output type, the activation (what selects the output computation), computation units, and the corresponding computation duration.

Output	Activation	Units	Computation Duration
MER	Default	Decibels	Per burst
Min MER	Parameter setting	Decibels	Per burst

Output	Activation	Units	Computation Duration
Percentile MER	Parameter setting	Decibels	Continuous
Number of symbols	Parameter setting if you select **Output X-percentile MER**	None	Continuous

Dimension

The block computes measurements for bursts of data. The data must be of length N symbols, where N is the size of the burst. The block computes a unique output for each incoming burst; therefore, the computation duration is per burst.

Input Signals

The input signals must be 1-D or 2-D sample-based column vectors or 2-D frame-based column vectors. The input and reference signals must have identical dimensions.

Output Signals

The output is always a scalar value.

Data Type

The block accepts double, single, and fixed-point data types. The output of the block is always double type.

Algorithms

MER is a measure of the SNR in a modulated signal calculated in dB. MER over N symbols is

$$MER = 10 * \log_{10}\left(\frac{\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{\sum_{n=1}^{N}(e_k)}\right) dB$$

The MER for the k_{th} symbol is

$$MER_k = 10 * \log_{10}\left(\frac{\frac{1}{N}\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{e_k}\right) db$$

The minimum MER represents the minimum MER value in a burst or

$$MER_{\min} = \min_{k \in [1,\ldots,N]} \{MER_k\}$$

where

$$e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$$

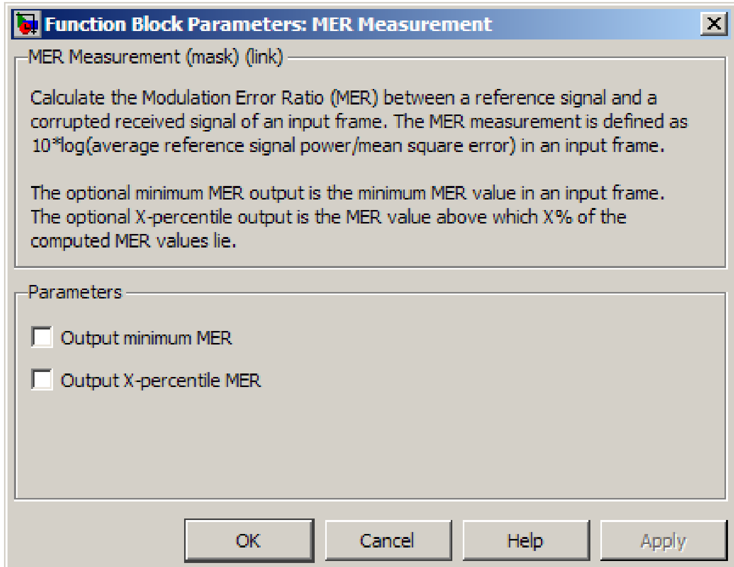
I_k = In-phase measurement of the *kth* symbol in the burst

Q_k = Quadrature phase measurement of the *kth* symbol in the burst

I_k and Q_k represent ideal (reference) values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbols.

The block computes X-percentile MER by creating a histogram of all the incoming MER_k values. The output provides the MER value above which X% of the MER values lay.

Dialog Box



Output Minimum MER
: Outputs the minimum MER of an input vector or frame.

Output X-percentile MER
: Enables an output X-percentile MER measurement. When you select this option, specify **X-percentile value (%)**.

X-Percentile value (%)
: This parameter only appears when you select **Output X-percentile MER**. The Xth percentile is the MER value above which X% of all the computed MER values lie. The parameter defaults to the 95th percentile. Therefore, 95% of all MER values are above this output.

Output the number of symbols processed
: Outputs the number of symbols that the block uses to compute the **Output X-percentile MER**. This parameter only appears when you select **Output X-percentile MER**.

See Also

EVM Measurement

References

[1] *Digital Video Broadcasting (DVB): Measurement guidelines for DVB systems*, DVB (ETSI) Standard ETR290, May 1997.

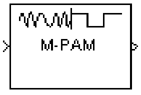
Purpose

Demodulate PAM-modulated data

Library

AM, in Digital Baseband sublibrary of Modulation

Description



The M-PAM Demodulator Baseband block demodulates a signal that was modulated using M-ary pulse amplitude modulation. The input is a baseband representation of the modulated signal.

The signal constellation has M points, where M is the **M-ary number** parameter. M must be an even integer. The block scales the signal constellation based on how you set the **Normalization method** parameter. For details on the constellation and its scaling, see the reference page for the M-PAM Modulator Baseband block.

This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see "Supported Data Types" on page 2-564.

Note All values of power assume a nominal impedance of 1 ohm.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Output type** parameter to `Integer`, the block outputs integer values between `0` and M-`1`. M represents the **M-ary number** block parameter.

When you set the **Output type** parameter to `Bit`, the block outputs binary-valued signals that represent integers. The block represents each integer using a group of $K = \log_2(M)$ bits, where K represents the number of bits per symbol. The output vector length must be an integer multiple of K.

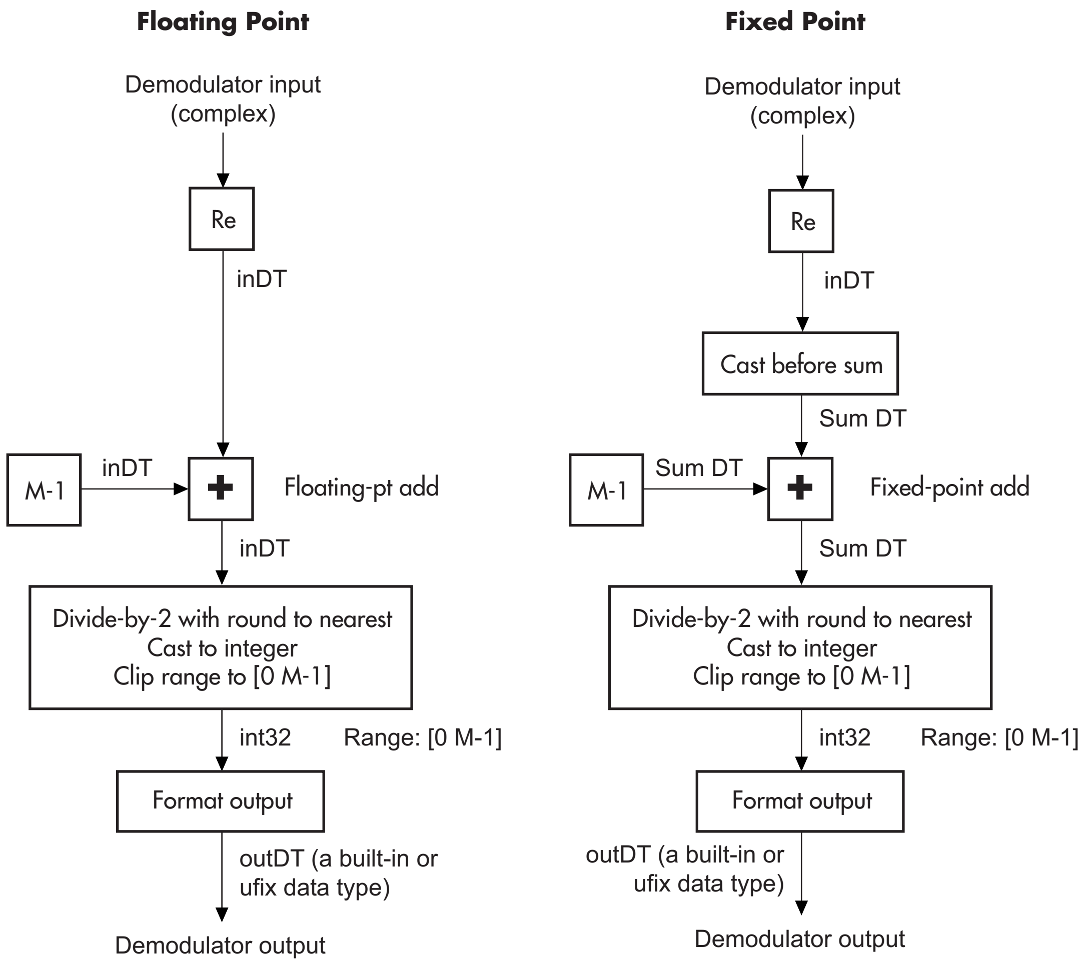
The **Constellation ordering** parameter indicates how the block assigns binary words to points of the signal constellation. More details are on the reference page for the M-PAM Modulator Baseband block.

Algorithm

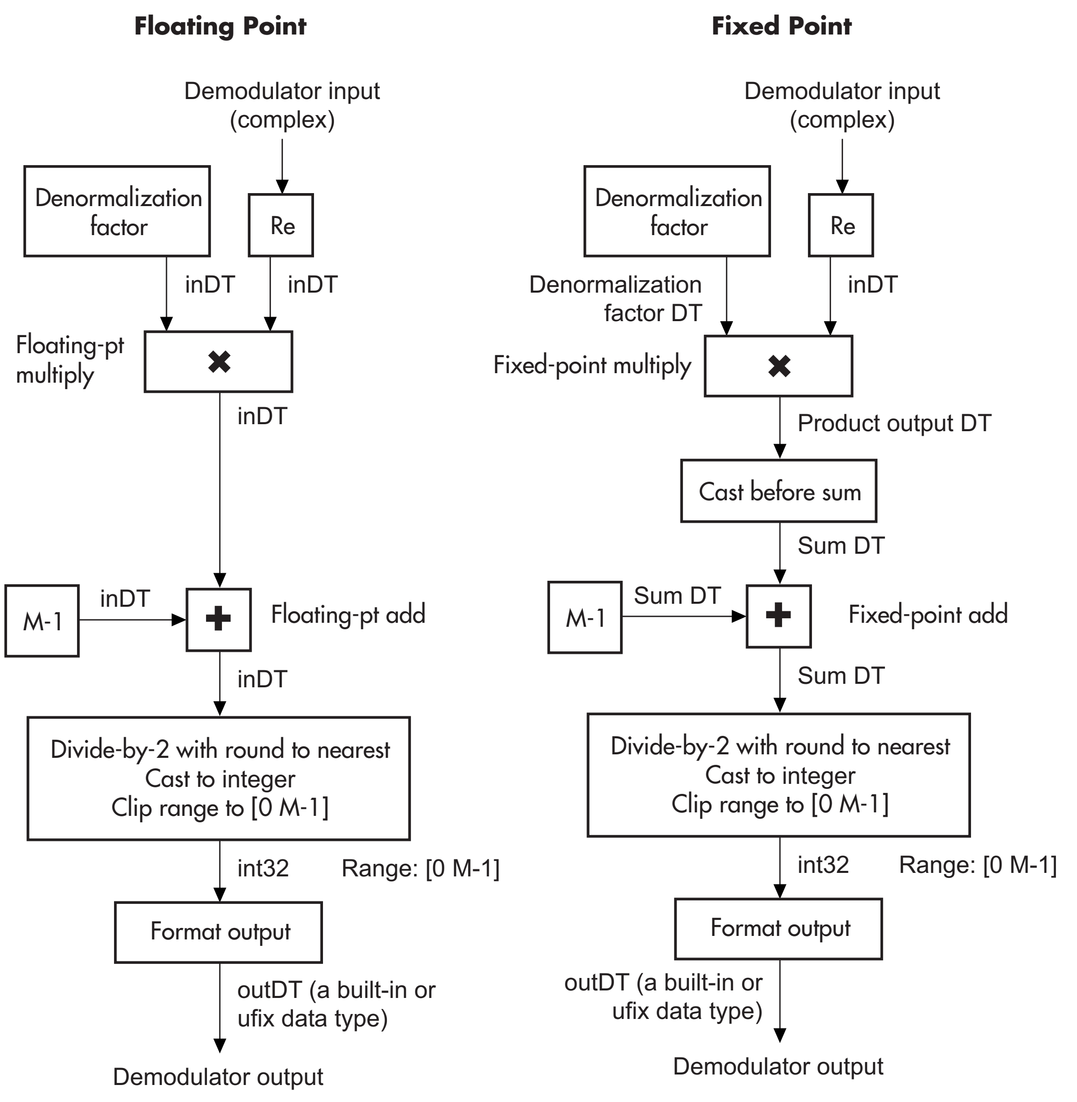
The demodulator algorithm maps received input signal constellation values to M-ary integer symbol indices between 0 and M-1 and then maps these demodulated symbol indices to formatted output values.

The integer symbol index computation is performed by first scaling the real part of the input signal constellation (possibly with noise) by a denormalization factor derived from the **Normalization method** and related parameters. This denormalized value is added to M-1 to translate it into an approximate range between 0 and 2 x (M-1) plus noise. The resulting value is then rescaled via a divide-by-two (or, equivalently, a right-shift by one bit for fixed-point operation) to obtain a range approximately between 0 and M-1 (plus noise). The noisy index value is rounded to the nearest integer and clipped, via saturation, to the exact range of [0 M-1]. Finally, based on other block parameters, the integer index is mapped to a symbol value that is formatted and cast to the selected **Output data type**.

The following figures contains signal flow diagrams for floating-point and fixed-point algorithm operation. The floating-point diagrams apply when the input signal data type is `double` or `single`. The fixed-point diagrams apply when the input signal is a signed fixed-point data type. Note that the diagram is simplified when using normalized constellations (i.e., denormalization factor is 1).

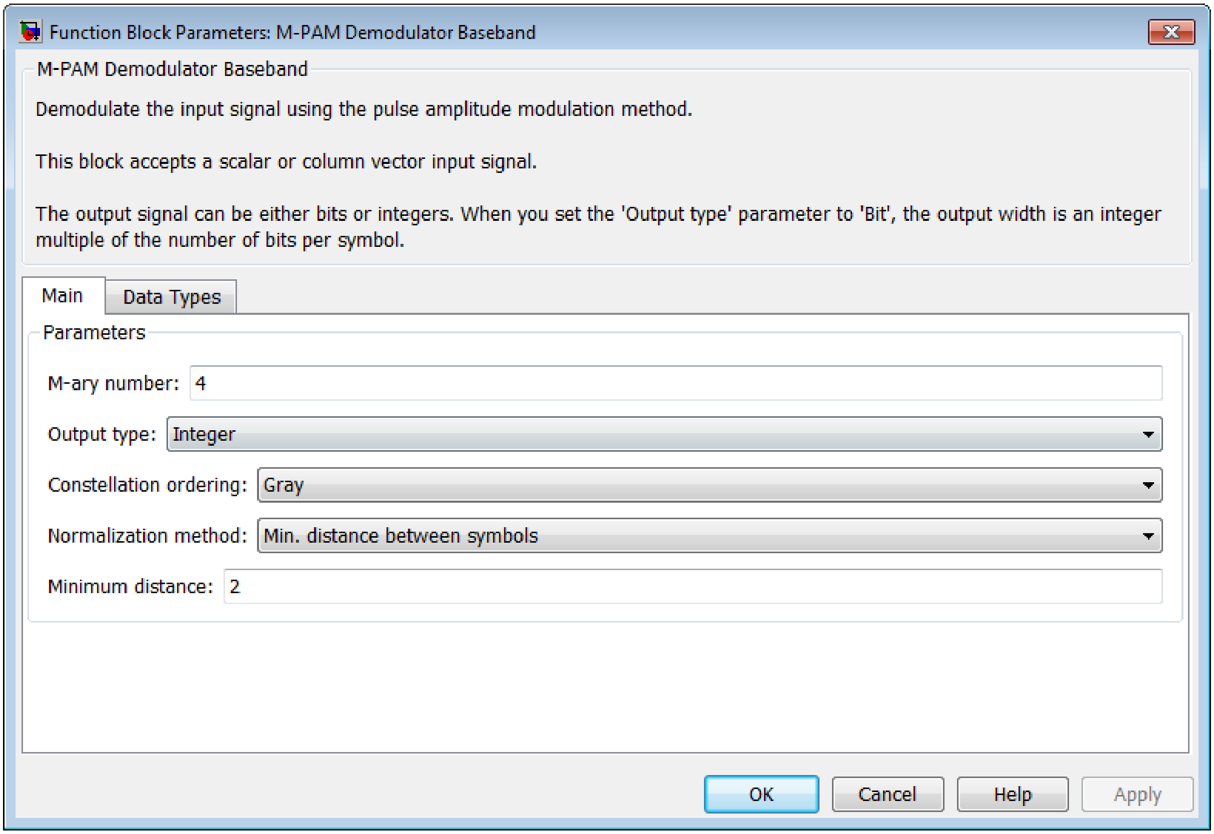


Signal-Flow Diagrams with Denormalization Factor Equal to 1



Signal-Flow Diagrams with Nonunity Denormalization Factor

Dialog Box



M-ary number

The number of points in the signal constellation. It must be an even integer.

Output type
: Determines whether the output consists of integers or groups of bits. If this parameter is set to `Bit`, then the **M-ary number** parameter must be 2^K for some positive integer K.

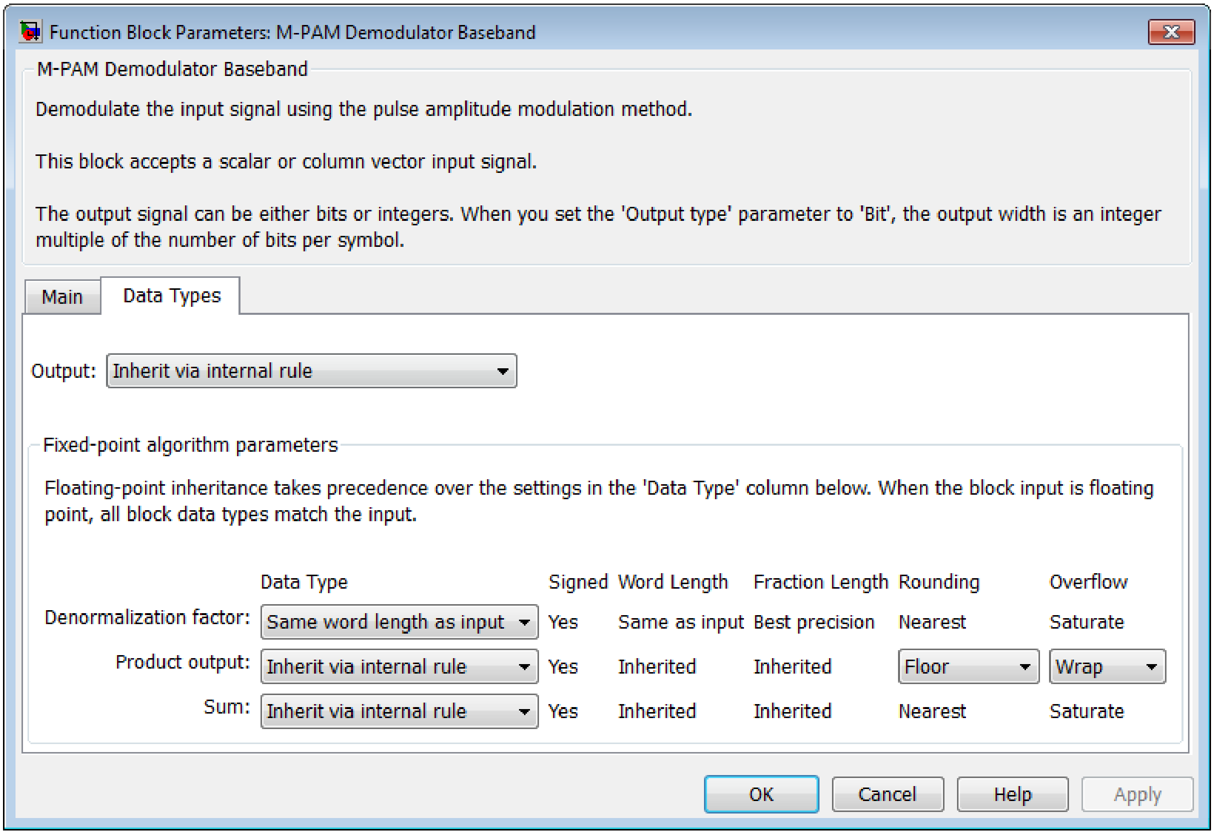
Constellation ordering
: Determines how the block maps each integer to a group of output bits.

Normalization method
: Determines how the block scales the signal constellation. Choices are `Min. distance between symbols`, `Average Power`, and `Peak Power`.

Minimum distance
: The distance between two nearest constellation points. This field appears only when **Normalization method** is set to `Min. distance between symbols`.

Average power, referenced to 1 ohm (watts)
: The average power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Average Power`.

Peak power, referenced to 1 ohm (watts)
: The maximum power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Peak Power`.



Output

When the parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is of type `single` or `double`. Otherwise, the output data type will be as if this parameter is set to `'Smallest unsigned integer'`.

When the parameter is set to `'Smallest unsigned integer'`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type is the ideal minimum size, i.e., `ufix(1)` for bit outputs, and `ufix(ceil(log2(M)))` for integer outputs. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit the ideal minimum size, usually corresponding to the size of a char (e.g., `uint8`).

For integer outputs, this parameter can be set to `Smallest unsigned integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`. For bit outputs, the options are `Smallest unsigned integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, or `double`.

Denormalization factor

This parameter applies when a fixed-point input is not normalized. It can be set to `Same word length as input` or `Specify word length`, in which case a field is enabled for user input. A best-precision fraction length is always used.

Product output

This parameter only applies when the input is a fixed-point signal and there is a nonunity (not equal to 1) denormalized factor. It can be set to `Inherit via internal rule` or `Specify word length`, which enables a field for user input.

Setting to `Inherit via internal rule` computes the full-precision product word length and fraction length. Internal Rule for Product Data Types in *DSP System Toolbox User's Guide* describes the full-precision Product output internal rule.

Setting to `Specify word length` allows you to define the word length. The block computes a best-precision fraction length based on the word length specified and the pre-computed worst-case (min/max) real world value **Product output** result. The worst-case **Product output** result is precomputed by multiplying the denormalized factor with the worst-case (min/max) input signal range, purely based on the input signal data type.

The block uses the **Rounding** method when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. For more information, see "Rounding Modes" in the DSP System Toolbox documentation or "Rounding Mode: Simplest" in the Fixed-Point Designer documentation.

Sum

This parameter only applies when the input is a fixed-point signal. It can be set to `Inherit via internal rule`, `Same as product output`, or `Specify word length`, in which case a field is enabled for user input

Setting `Inherit via internal rule` computes the full-precision sum word length and fraction length, based on the two inputs to the Sum in the fixed-point Hard Decision Algorithm signal flow diagram. The rule is the same as the fixed-point inherit rule of the internal **Accumulator data type** parameter in the Simulink Sum block.

Setting `Specify word length` allows you to define the word length. A best precision fraction length is computed based on the word length specified in the pre-computed maximum range necessary for the demodulated algorithm to produce accurate results. The signed fixed-point data type that has the best precision fully contains the values in the range 2 * (M-1) for the specified word length.

Setting to `Same as product output` allows the Sum data type to be the same as the **Product output** data type (when **Product output** is used). If the **Product output** is not used, then this setting will be ignored and the `Inherit via internal rule` Sum setting will be used.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed-point
Output	• Double-precision floating point • Single-precision floating point • Boolean when **Output type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) in ASIC/FPGA when **Output type** is `Bit` • $ufix(\lceil \log_2 M \rceil)$ in ASIC/FPGA when **Output type** is `Integer`

Pair Block

M-PAM Modulator Baseband

See Also

General QAM Demodulator Baseband

M-PAM Modulator Baseband

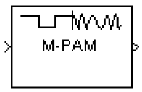
Purpose

Modulate using M-ary pulse amplitude modulation

Library

AM, in Digital Baseband sublibrary of Modulation

Description



The M-PAM Modulator Baseband block modulates using M-ary pulse amplitude modulation. The output is a baseband representation of the modulated signal. The **M-ary number** parameter, M, is the number of points in the signal constellation. It must be an even integer.

Note All values of power assume a nominal impedance of 1 ohm.

Constellation Size and Scaling

Baseband M-ary pulse amplitude modulation using the block's default signal constellation maps an integer m between 0 and M-1 to the complex value

$$2m - M + 1$$

Note This value is actually a real number. The block's output signal is a complex data-type signal whose imaginary part is zero.

The block scales the default signal constellation based on how you set the **Normalization method** parameter. The following table lists the possible scaling conditions.

Value of Normalization Method Parameter	Scaling Condition
`Min. distance between symbols`	The nearest pair of points in the constellation is separated by the value of the **Minimum distance** parameter
`Average Power`	The average power of the symbols in the constellation is the **Average power** parameter
`Peak Power`	The maximum power of the symbols in the constellation is the **Peak power** parameter

Integer-Valued Signals and Binary-Valued Signals

This block accepts a scalar or column vector input signal.

When you set the **Input type** parameter to `Integer`, the block accepts integer values between `0` and M-`1`. M represents the **M-ary number** block parameter.

When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

where

K represents the number of bits per symbol.

The input vector length must be an integer multiple of K. In this configuration, the block accepts a group of K bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol for each group of K bits.

The **Constellation ordering** parameter indicates how the block assigns binary words to points of the signal constellation.

- If **Constellation ordering** is set to `Binary`, then the block uses a natural binary-coded constellation.

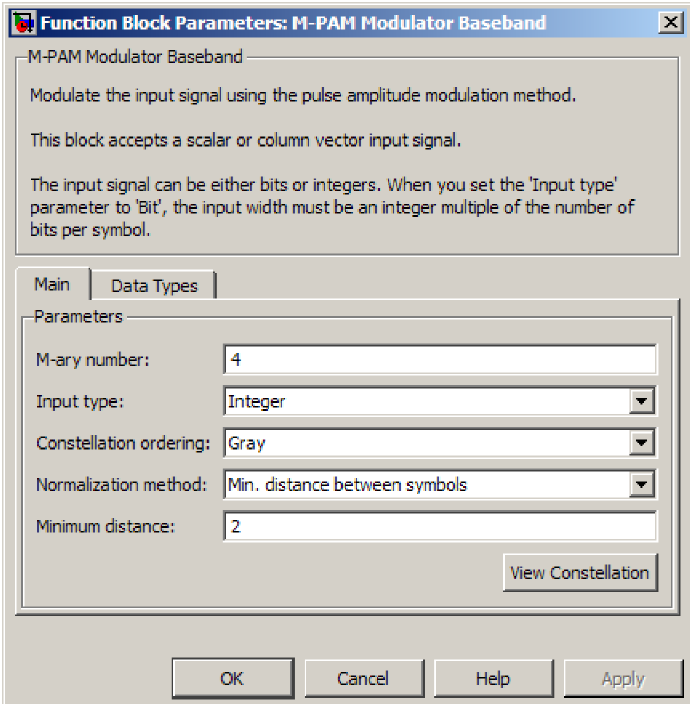
- If **Constellation ordering** is set to `Gray`, then the block uses a Gray-coded constellation.

  For details about the Gray coding, see the reference page for the M-PSK Modulator Baseband block.

Constellation Visualization

The M-PAM Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. For more information, see the Constellation Visualization section of the *Communications System Toolbox User's Guide*.

Dialog Box



M-ary number

The number of points in the signal constellation. It must be an even integer.

Input type

Indicates whether the input consists of integers or groups of bits. If this parameter is set to `Bit`, then the **M-ary number** parameter must be 2^K for some positive integer K.

Constellation ordering

Determines how the block maps each group of input bits to a corresponding integer.

Normalization method
: Determines how the block scales the signal constellation. Choices are `Min. distance between symbols`, `Average Power`, and `Peak Power`.

Minimum distance
: The distance between two nearest constellation points. This field appears only when **Normalization method** is set to `Min. distance between symbols`.

Average power, referenced to 1 ohm (watts)
: The average power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Average Power`.

Peak power, referenced to 1 ohm (watts)
: The maximum power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Peak Power`.

Output data type
: The output data type can be set to `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

  Setting this parameter to `Fixed-point` or `User-defined` enables fields in which you can further specify details. Setting this parameter to `Inherit via back propagation`, sets the output data type and scaling to match the following block.

Output 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 signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer software. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set output fraction length to

Specify the scaling of the fixed-point output by either of the following 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 **Output 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.

Output 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 output fraction length to** parameter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean when **Input type** is `Bit` • 8-, 16-, 32-bit signed integers • 8-, 16-, 32-bit unsigned integers

Port	Supported Data Types
	• $ufix(\lceil \log_2 M \rceil)$ when **Input type** is `Integer`
Output	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

M-PAM Demodulator Baseband

See Also

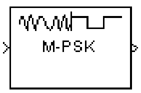
General QAM Modulator Baseband

M-PSK Demodulator Baseband

Purpose Demodulate PSK-modulated data

Library PM, in Digital Baseband sublibrary of Modulation

Description The M-PSK Demodulator Baseband block demodulates a signal that was modulated using the M-ary phase shift keying method. The input is a baseband representation of the modulated signal. The input and output for this block are discrete-time signals. This block accepts a scalar-valued or column vector input signal. For information about the data types each block port supports, see “Supported Data Types” on page 2-586.



The **M-ary number** parameter, M, is the number of points in the signal constellation.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Output type** parameter to `Integer`, the block outputs integer values between `0` and M-`1`. M represents the **M-ary number** block parameter.

When you set the **Output type** parameter to `Bit`, the block outputs binary-valued signals that represent integers. The block represents each integer using a group of $K = \log_2(M)$ bits, where K represents the number of bits per symbol. The output vector length must be an integer multiple of K.

Depending on the demodulation scheme, the **Constellation ordering** or **Symbol set ordering** parameter indicates how the block maps a symbol to a group of K output bits. When you set the parameter to `Binary`, the block maps the integer, I, to [u(1) u(2) ... u(K)] bits, where the individual u(1) are given by

$$\sum_{i=1}^{K} u(i) 2^{K-i}$$

u(1) is the most significant bit.

For example, if $M = 8$, **Constellation ordering** (or **Symbol set ordering**) is set to `Binary`, and the integer symbol value is 6, then the binary input word is [1 1 0].

When you set **Constellation ordering** (or **Symbol set ordering**) to `Gray`, the block assigns binary outputs from points of a predefined Gray-coded signal constellation. The predefined M-ary Gray-coded signal constellation assigns the binary representation

```
de2bi(bitxor(M,floor(M/2)), log2(M),'left-msb')
```

to the M^{th} phase. The zeroth phase in the constellation is the **Phase offset** parameter, and successive phases are counted in a counterclockwise direction.

Note This transformation might seem counterintuitive because it constitutes a Gray-to-binary mapping. However, the block must use it to impose a Gray ordering on the signal constellation, which has a natural binary ordering.

In other words, if the block input is the natural binary representation, u, of the integer U, the block output has phase

$$j\theta + j2\pi m/M$$

where θ is the **Phase offset** parameter and m is an integer between 0 and M-1 that satisfies

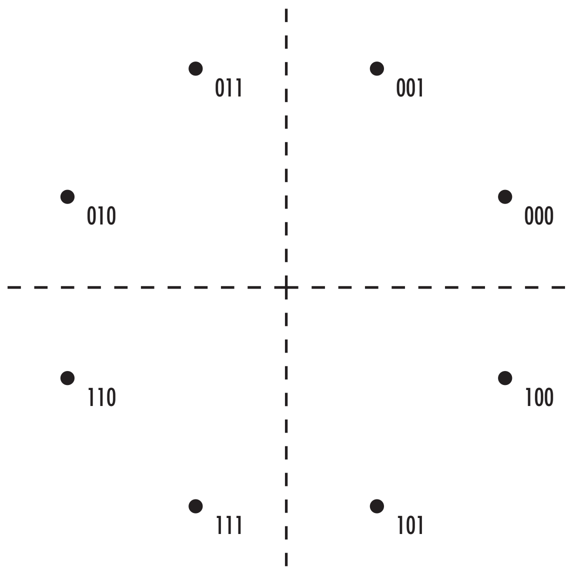
$$m \text{ XOR } \lfloor m/2 \rfloor = U$$

For example, if M = 8, the binary representations that correspond to the zeroth through seventh phases are as follows.

```
M = 8; m = [0:M-1]';
de2bi(bitxor(m,floor(m/2)), log2(M),'left-msb')
```

```
ans =

     0     0     0
     0     0     1
     0     1     1
     0     1     0
     1     1     0
     1     1     1
     1     0     1
     1     0     0
```

The following diagram shows the 8-ary Gray-coded constellation that the block uses if the **Phase offset** parameter is $\frac{\Pi}{8}$.



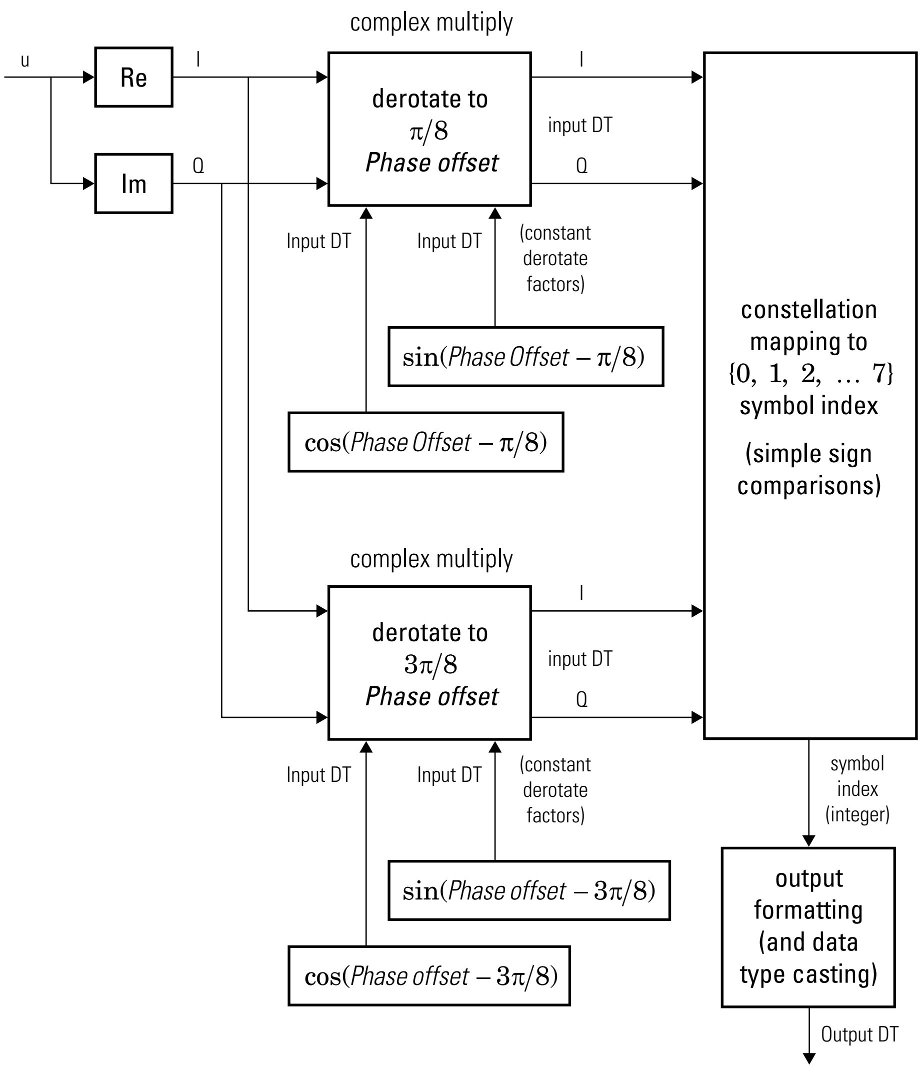
.

Algorithm

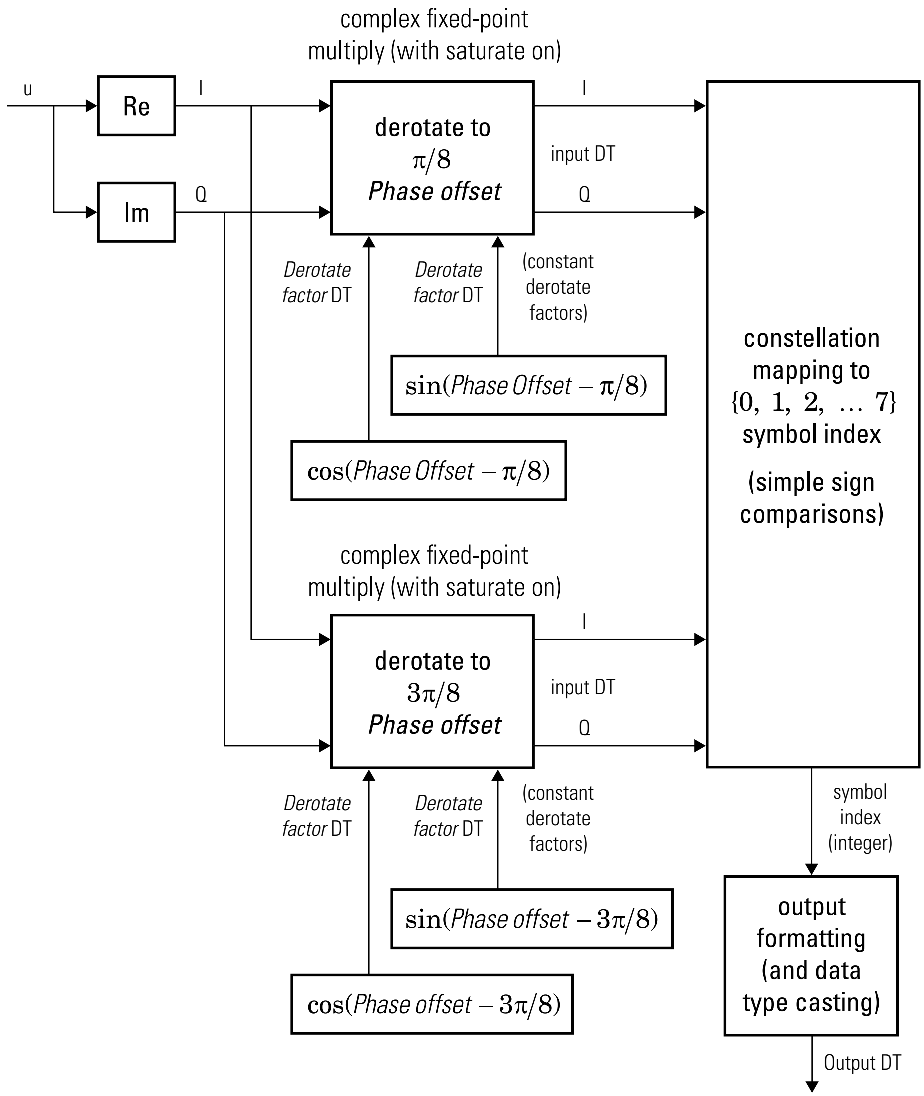
For M=2, refer to the BPSK Demodulator Baseband block reference page.

For M=4, refer to the QPSK Demodulator Baseband block reference page.

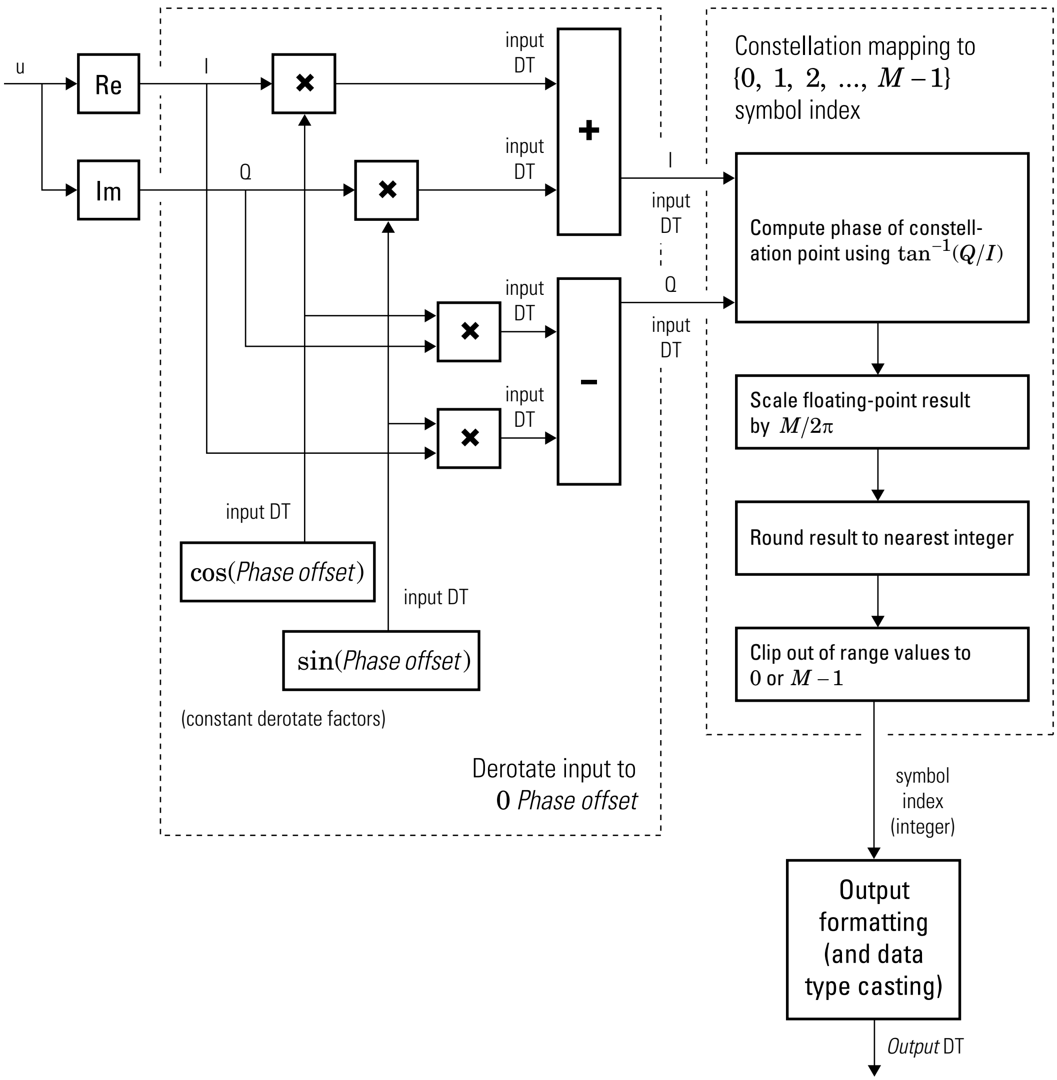
For M=8 and greater, see the following signal diagrams.



Hard-Decision 8-PSK Demodulator Floating-Point Signal Diagram



Hard-Decision 8-PSK Demodulator Fixed-Point Signal Diagram



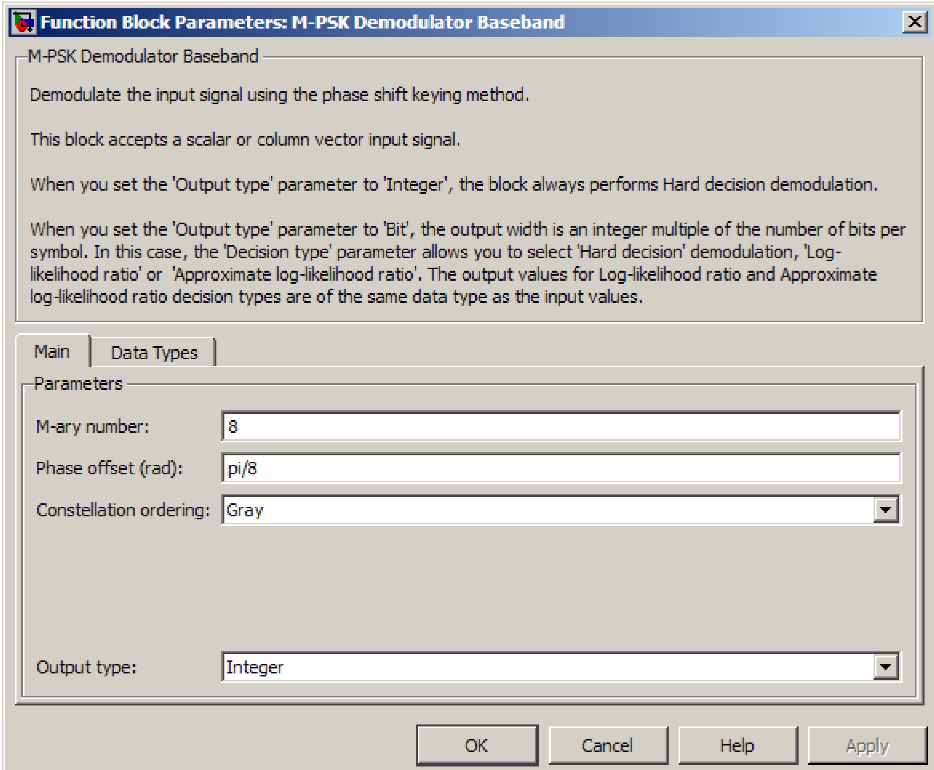
Hard-Decision M-PSK Demodulator (M > 8) Floating-Point Signal Diagram for Nontrivial Phase Offset

For $M > 8$, in order to improve speed and implementation costs, no derotation arithmetic is performed when **Phase offset** is 0, $\pi/2$, π, or $3\pi/2$ (i.e., when it is trivial).

Also, for $M > 8$, this block will only support inputs of type `double` and `single`.

The exact LLR and approximate LLR cases (soft-decision) are described in "Exact LLR Algorithm" and "Approximate LLR Algorithm" in the *Communications System Toolbox User's Guide*.

Dialog Box



M-ary number

The number of points in the signal constellation.

Phase offset

The phase of the zeroth point of the signal constellation.

Constellation ordering

Determines how the block maps a symbol to the corresponding K output bits or integer. See the reference page for the M-PSK Modulator Baseband block for details. Selecting `User-defined` displays the field **Constellation mapping**, allowing for user-specified mapping.

Constellation mapping

This field appears when `User-defined` is selected in the drop-down list **Constellation ordering**.

This parameter is a row or column vector of size M and must have unique integer values in the range [0, M-1]. The values must be of data type `double`.

The first element of this vector corresponds to the constellation point at 0 + **Phase offset** angle, with subsequent elements running counterclockwise. The last element corresponds to the $-2\pi/M$ + **Phase offset** constellation point.

Output type

Determines whether the output consists of integers or groups of bits. If this parameter is set to `Bit`, the **M-ary number** parameter must be 2^K for some positive integer K.

Decision type

Specifies the output to be bitwise hard decision, LLR, or approximate LLR. This parameter appears when you select `Bit` from the **Output type** drop-down list. The output values for Log-likelihood ratio and Approximate log-likelihood ratio decision types are of the same data type as the input values

See "Exact LLR Algorithm" and "Approximate LLR Algorithm" in the *Communications System Toolbox User's Guide* for algorithm details.

Noise variance source

This field appears when `Approximate log-likelihood ratio` or `Log-likelihood ratio` is selected for **Decision type**.

When set to `Dialog`, the noise variance can be specified in the **Noise variance** field. When set to `Port`, a port appears on the block through which the noise variance can be input.

Noise variance

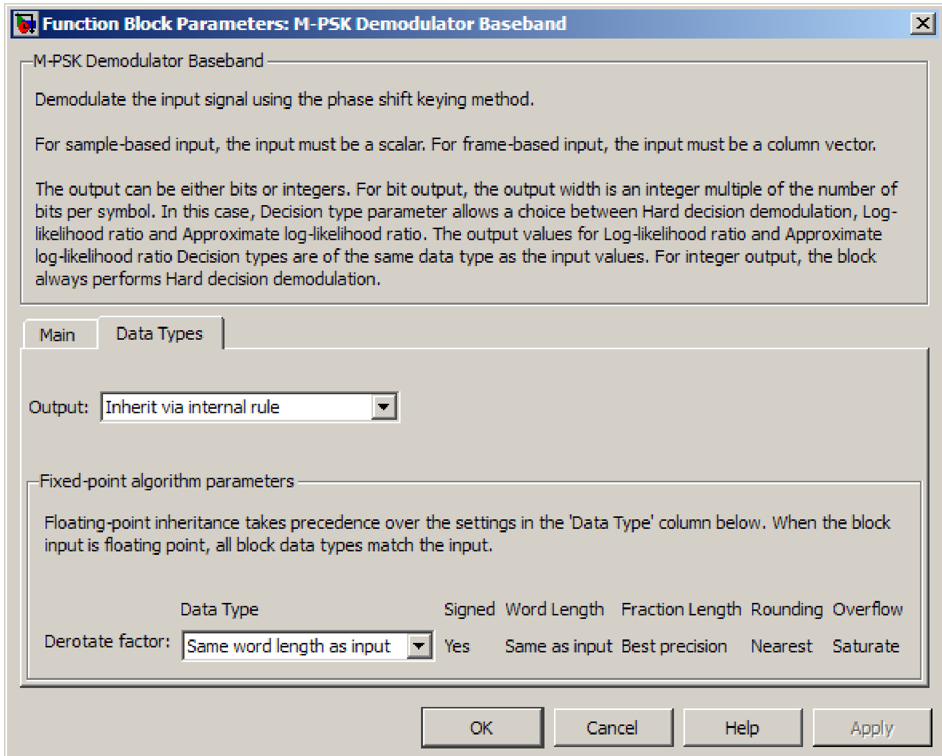
This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

If you use theSimulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

The LLR algorithm involves computing exponentials of very large or very small numbers using finite precision arithmetic and would yield:

- `Inf` to `-Inf` if **Noise variance** is very high
- `NaN` if **Noise variance** and signal power are both very small

In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.



Data Types Pane for Hard-Decision

Output

For bit outputs, when **Decision type** is set to `Hard decision`, the output data type can be set to `'Inherit via internal rule'`, `'Smallest unsigned integer'`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `boolean`.

For integer outputs, the output data type can be set to `'Inherit via internal rule'`, `'Smallest unsigned integer'`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

When this parameter is set to 'Inherit via internal rule' (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is a floating-point type (single or double). If the input data type is fixed-point (supported only when **M-ary number** is 2, 4, or 8), the output data type will work as if this parameter is set to 'Smallest unsigned integer'.

When this parameter is set to 'Smallest unsigned integer', the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model.

If ASIC/FPGA is selected in the **Hardware Implementation** pane, and **Output type** is Bit, the output data type is the ideal minimum one-bit size, i.e., ufix(1). For all other selections, it is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding to the size of a char (e.g., uint8).

If ASIC/FPGA is selected in the **Hardware Implementation** pane, and **Output type** is Integer, the output data type is the ideal minimum integer size, i.e., ufix(ceil(log2(M))). For all other selections, it is an unsigned integer with the smallest available word length large enough to fit the ideal minimum size, usually corresponding to the size of a char (e.g., uint8).

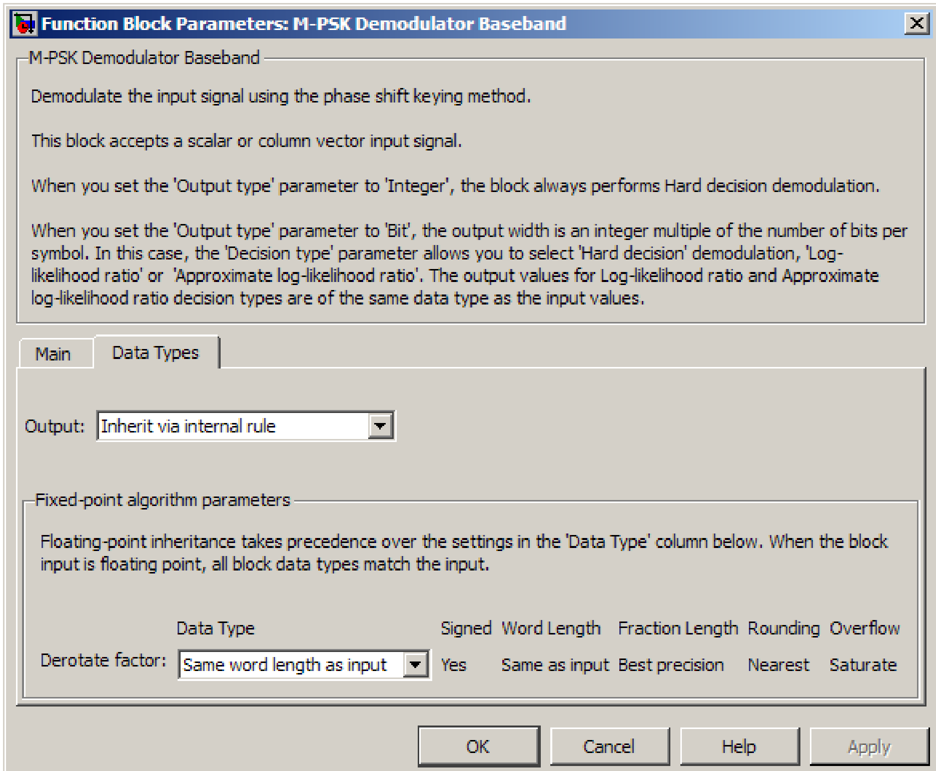
Derotate factor

This parameter only applies when **M-ary number** is 2, 4, or 8, the input is fixed-point, and **Phase offset** is nontrivial. The phase offset is trivial when:

- You set **M-ary number** to 2 and **Phase offset** to a multiple of $\frac{\pi}{2}$

- You set **M-ary number** to 4 and **Phase offset** to an even multiple of $\frac{\pi}{4}$

When you set **M-ary number** to 8 there are no trivial phase offsets.



Data Types Pane for Soft-Decision

For bit outputs, when **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the output data type is inherited from the input (e.g., if the input is of data type `double`, the output is also of data type `double`).

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed point only when $M \leq 8$ and: - **Output type** is `Integer` - **Output type** is `Bit` and **Decision type** is `Hard-decision`
Var	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point • Boolean when **Output type** is `Bit` and **Decision type** is `Hard-decision` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) when **Output type** is `Bit` • $ufix(\log_2 M)$ when **Output type** is `Integer`

Pair Block

M-PSK Modulator Baseband

See Also

BPSK Demodulator Baseband, QPSK Demodulator Baseband, M-DPSK Demodulator Baseband

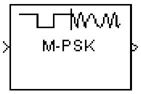
Purpose

Modulate using M-ary phase shift keying method

Library

PM, in Digital Baseband sublibrary of Modulation

Description



The M-PSK Modulator Baseband block modulates using the M-ary phase shift keying method. The output is a baseband representation of the modulated signal. The **M-ary number** parameter, M, is the number of points in the signal constellation.

The block accepts scalar or column vector input signals. For information about the data types each block port supports, see "Supported Data Types" on page 2-594.

Alternative configurations of the block determine how the block interprets its input and arranges its output, as explained in the following sections.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, the block accepts integer values between `0` and M-`1`. M represents the **M-ary number** block parameter.

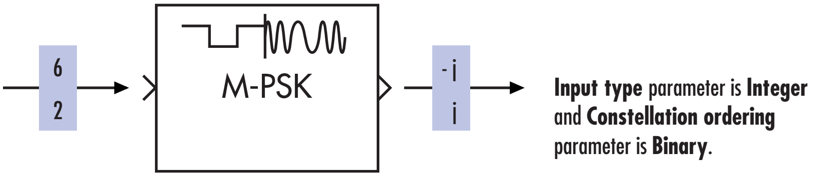
When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

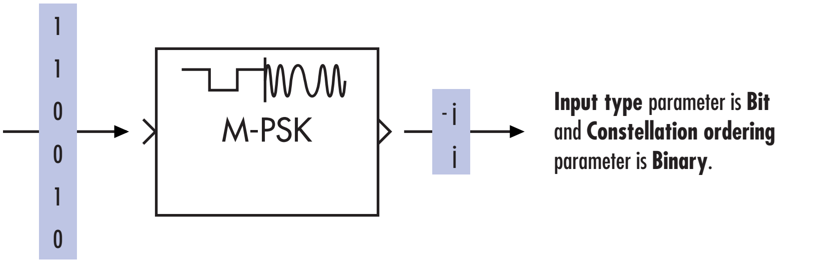
where

K represents the number of bits per symbol.

The input vector length must be an integer multiple of K. In this configuration, the block accepts a group of K bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol for each group of K bits.

For example, the following schematics illustrate how the block processes two 8-ary integers or binary words in one time step. The block processes all input signals as frames. In both cases, the **Phase offset** parameter is `0`.





The **Constellation ordering** parameter indicates how the block maps a group of K input bits to a corresponding symbol. When you set the parameter to `Binary`, the block maps [u(1) u(2) ... u(K)] to the integer

$$\sum_{i=1}^{K} u(i)2^{K-i}$$

and behaves as if this integer were the input value. u(1) is the most significant bit.

For example, if you set M = 8, **Constellation ordering** to `Binary`, and the binary input word is [1 1 0], the block converts [1 1 0] to the integer 6. The block produces the same output when the input is 6 and the **Input type** parameter is `Integer`.

When you set **Constellation ordering** to `Gray`, the block uses a Gray-coded arrangement and assigns binary inputs to points of a predefined Gray-coded signal constellation. The predefined M-ary Gray-coded signal constellation assigns the binary representation

```
de2bi(bitxor(M,floor(M/2)), log2(M),'left-msb')
```

to the M^{th} phase. The zeroth phase in the constellation is the **Phase offset** parameter. Successive phases are in the counterclockwise direction.

Note This transformation seems counterintuitive because it constitutes a Gray-to-binary mapping. However, the block must use it to impose a Gray ordering on the signal constellation, which has a natural binary ordering.

In other words, if the block input is the natural binary representation, u, of the integer U, the block output has phase

$$j\theta + j2\pi m/M$$

where θ is the **Phase offset** parameter and m is an integer between 0 and M-1 that satisfies

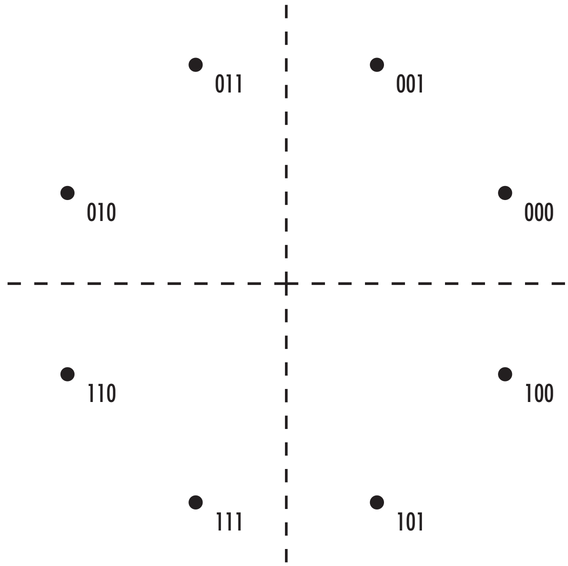
$$m \text{ XOR } \lfloor m/2 \rfloor = U$$

For example, if M = 8, the binary representations that correspond to the zeroth through seventh phases are as follows.

```
M = 8; m = [0:M-1]';
de2bi(bitxor(m,floor(m/2)), log2(M),'left-msb')

ans =

     0     0     0
     0     0     1
     0     1     1
     0     1     0
     1     1     0
     1     1     1
```

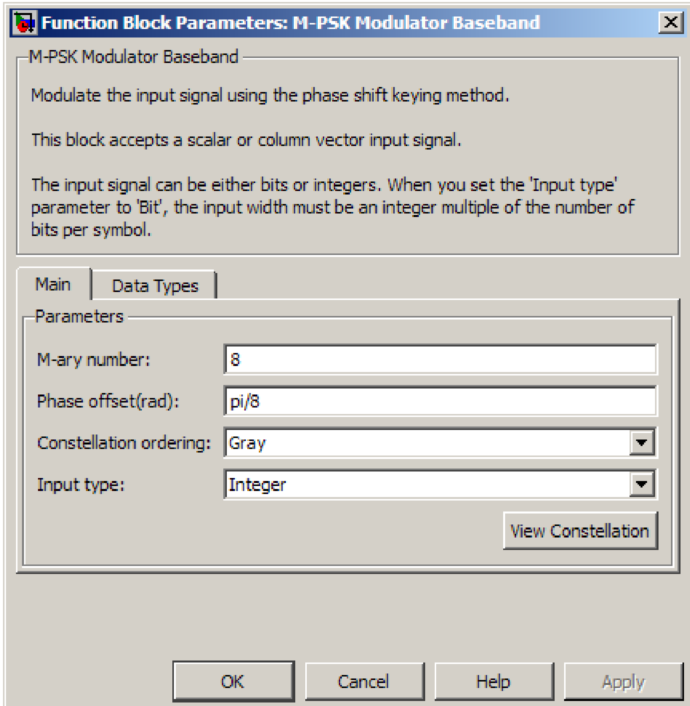
1	0	1
1	0	0

The following diagram shows the 8-ary Gray-coded constellation that the block uses if the **Phase offset** parameter is $\frac{\Pi}{8}$.



Constellation Visualization

The M-PSK Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. For more information, see the Constellation Visualization section of the *Communications System Toolbox User's Guide*.



Dialog Box

M-ary number

The number of points in the signal constellation.

Phase offset

The phase of the zeroth point of the signal constellation.

Constellation ordering

Determines how the block maps an integer or group of K input bits to the corresponding symbol.

If set to `Binary`, baseband M-ary phase shift keying modulation with a phase offset of θ maps an integer *m* between 0 and M-1 to the complex value exp(jθ + j2πm/M)

If set to `Gray`, the block uses a Gray-coded signal constellation. As a result, binary representations that differ in more than one bit cannot map to consecutive integers modulo M.

Selecting `User-defined` displays the **Constellation mapping** parameter, which allows you to specify the mapping technique for the block.

Constellation mapping

This field appears when you select `User-defined` from the **Constellation ordering** drop-down list.

This parameter is a row or column vector of size M and must have unique integer values in the range [0, M-1]. The values must be of data type `double`.

The first element of this vector corresponds to the constellation point at 0 + **Phase offset** angle, with subsequent elements running counterclockwise. The last element corresponds to the -2π/M + **Phase offset** constellation point.

Input type

Indicates whether the input consists of integers or groups of bits.

To use integer values between `0` and M-1 as inputs, set this parameter to `Integer`.

If this parameter is set to `Bit`, the **M-ary number** parameter must be 2^K for some positive integer K. K consecutive elements in the input represent a symbol, where K = log2(M).

Output data type

This block supports the following output data types: `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

Set this property to `Fixed-point` or `User-defined` to enable parameters in which you specify additional details. Set this property to `Inherit via back propagation` to match the output data type and scaling to the following block in the model.

Output word length

Specify the word length, in bits, of the fixed-point output data type. This parameter appears when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

Specify any signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer software. This parameter appears when you select `User-defined` for the **Output data type** parameter.

Output fraction length

Specify the scaling of the fixed-point output by either of the following methods:

- Select `Best precision` to automatically scale the output signal so that it has the best possible precision.
- Select `User-defined` to specify the output scaling in the **Output fraction length** parameter.

This parameter appears 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.

Output 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 appears when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set output fraction length to** parameter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean when **Input type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) when **Input type** is `Bit` • $ufix(\log_2 M)$ when **Input type** is `Integer`
Output	• Double-precision floating point • Single-precision floating point • Signed Fixed point

Pair Block

M-PSK Demodulator Baseband

See Also

BPSK Modulator Baseband, QPSK Modulator Baseband, M-DPSK Modulator Baseband

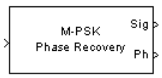
Purpose

Recover carrier phase using M-Power method

Library

Carrier Phase Recovery sublibrary of Synchronization

Description



The M-PSK Phase Recovery block recovers the carrier phase of the input signal using the M-Power method. This feedforward, non-data-aided, clock-aided method is suitable for systems that use baseband phase shift keying (PSK) modulation. It is also suitable for systems that use baseband quadrature amplitude modulation (QAM), although the results are less accurate than those for comparable PSK systems. The alphabet size for the modulation must be an even integer.

For PSK signals, the **M-ary number** parameter represents the alphabet size. For QAM signals, the **M-ary number** should be 4 regardless of the alphabet size because the 4-power method is the most appropriate for QAM signals.

The M-Power method assumes that the carrier phase is constant over a series of consecutive symbols, and returns an estimate of the carrier phase for the series. The **Observation interval** parameter is the number of symbols for which the carrier phase is assumed constant. This number must be an integer multiple of the input signal's vector length.

Input and Output Signals

This block accepts a scalar or column vector input signal of type `double` or `single`. The input signal represents a baseband signal at the symbol rate, so it must be complex-valued and must contain one sample per symbol.

The outputs are as follows:

- The output port labeled `Sig` gives the result of rotating the input signal counterclockwise, where the amount of rotation equals the carrier phase estimate. The `Sig` output is thus a corrected version of the input signal, and has the same sample time and vector size as the input signal.

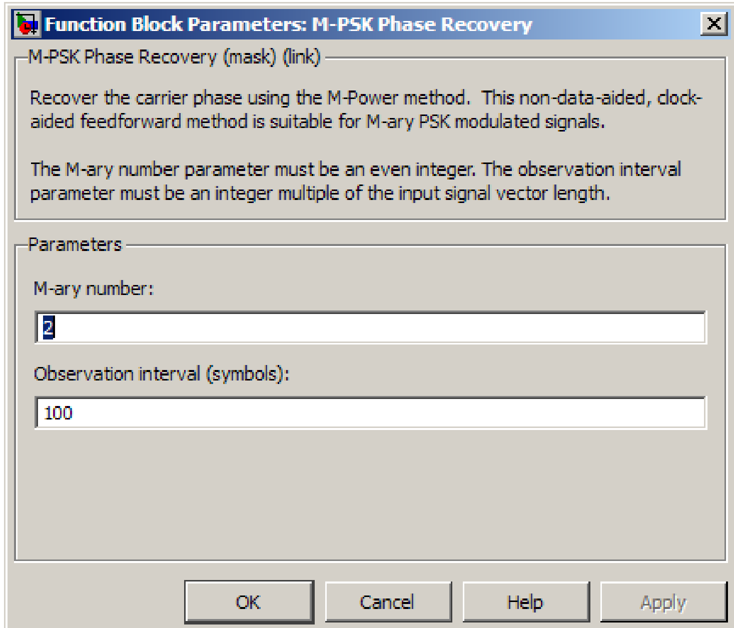
- The output port labeled `Ph` outputs the carrier phase estimate, in degrees, for all symbols in the observation interval. The `Ph` output is a scalar signal.

Note Because the block internally computes the argument of a complex number, the carrier phase estimate has an inherent ambiguity. The carrier phase estimate is between -180/M and 180/M degrees and might differ from the actual carrier phase by an integer multiple of 360/M degrees.

Delays and Latency

The block's algorithm requires it to collect symbols during a period of length **Observation interval** before computing a single estimate of the carrier phase. Therefore, each estimate is delayed by **Observation interval** symbols and the corrected signal has a latency of **Observation interval** symbols, relative to the input signal.

Dialog Box



M-ary number

The number of points in the signal constellation of the transmitted PSK signal. This value as an even integer.

Observation interval

The number of symbols for which the carrier phase is assumed constant. The observation interval parameter must be an integer multiple of the input signal vector length.

When this parameter is exactly equal to the vector length of the input signal, then the block always works. When the integer multiple is not equal to `1`, select **Simulation > Configuration Parameters > Solver**

and set **Tasking mode for periodic sample times** to `SingleTasking`.

Examples

See “Carrier Phase Recovery Example” in *Communications System Toolbox User’s Guide*.

Algorithm

If the symbols occurring during the observation interval are x(1), x(2), x(3),..., x(L), then the resulting carrier phase estimate is

$$\frac{1}{M}\arg\left\{\sum_{k=1}^{L}(x(k))^M\right\}$$

where the arg function returns values between -180 degrees and 180 degrees.

References

[1] Mengali, Umberto, and Aldo N. D’Andrea, *Synchronization Techniques for Digital Receivers*, New York, Plenum Press, 1997.

[2] Moeneclaey, Marc, and Geert de Jonghe, "ML-Oriented NDA Carrier Synchronization for General Rotationally Symmetric Signal Constellations," *IEEE Transactions on Communications*, Vol. 42, No. 8, Aug. 1994, pp. 2531-2533.

See Also

CPM Phase Recovery, M-PSK Modulator Baseband

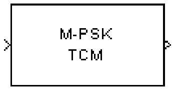
Purpose

Decode trellis-coded modulation data, modulated using PSK method

Library

TCM, in Digital Baseband sublibrary of Modulation

Description



The M-PSK TCM Decoder block uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using a PSK signal constellation.

The **M-ary number** parameter represents the number of points in the signal constellation, which also equals the number of possible output symbols from the convolutional encoder. (That is, $\log_2$(**M-ary number**) is the number of output bit streams from the convolutional encoder.)

The **Trellis structure** and **M-ary number** parameters in this block should match those in the M-PSK TCM Encoder block, to ensure proper decoding.

Input and Output Signals

This block accepts a column vector input signal containing complex numbers. The input signal must be `double` or `single`. The reset port signal must be `double` or `Boolean`. For information about the data types each block port supports, see "Supported Data Types" on page 2-602.

If the convolutional encoder described by the trellis structure represents a rate *k*/*n* code, then the M-PSK TCM Decoder block's output is a binary column vector whose length is *k* times the vector length of the input signal.

Operation Modes

The block has three possible methods for transitioning between successive frames. The **Operation mode** parameter controls which method the block uses. This parameter also affects the range of possible values for the **Traceback depth** parameter, D.

- In `Continuous` mode, the block initializes all state metrics to zero at the beginning of the simulation, waits until it accumulates D symbols, and then uses a sequence of D symbols to compute each of the traceback paths. D can be any positive integer. At the end of each frame, the block saves its internal state metric for use with the next frame.

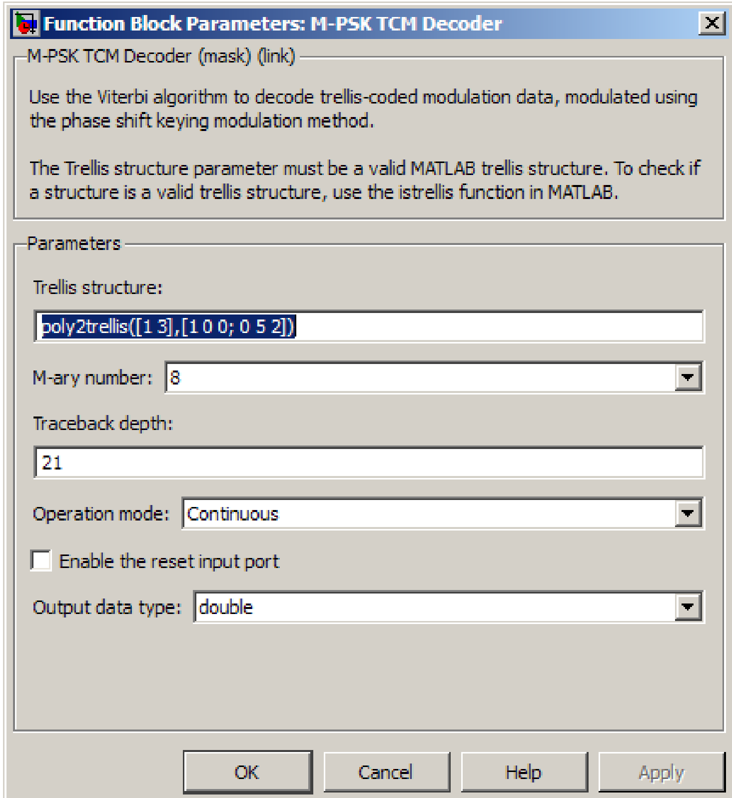
If you select **Enable the reset input**, the block displays another input port, labeled `Rst`. This port receives an integer scalar signal. Whenever the value at the `Rst` port is nonzero, the block resets all state metrics to zero and sets the traceback memory to zero.

- In `Truncated` mode, the block treats each frame independently. The traceback path starts at the state with the lowest metric. D must be less than or equal to the vector length of the input.
- In `Terminated` mode, the block treats each frame independently. The traceback path always starts at the all-zeros state. D must be less than or equal to the vector length of the input. If you know that each frame of data typically ends at the all-zeros state, then this mode is an appropriate choice.

Decoding Delay

If you set **Operation mode** to `Continuous`, then this block introduces a decoding delay equal to **Traceback depth***k bits, for a rate k/n convolutional code. The decoding delay is the number of zeros that precede the first decoded bit in the output.

The block incurs no delay for other values of **Operation mode**.



Dialog Box

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

M-ary number

The number of points in the signal constellation.

Traceback depth

The number of trellis branches (equivalently, the number of symbols) the block uses in the Viterbi algorithm to construct each traceback path.

Operation mode

The operation mode of the Viterbi decoder. Choices are `Continuous`, `Truncated`, and `Terminated`.

Enable the reset input port

When you check this box, the block has a second input port labeled `Rst`. Providing a nonzero input value to this port causes the block to set its internal memory to the initial state before processing the input data. This option appears only if you set **Operation mode** to `Continuous`.

Output data type

The output type of the block can be specified as a `boolean` or `double`. By default, the block sets this to `double`.

Supported Data Types

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

Pair Block

M-PSK TCM Encoder

See Also

General TCM Decoder, `poly2trellis`

References

[1] Biglieri, E., D. Divsalar, P. J. McLane and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

[2] Proakis, John G., *Digital Communications*, Fourth edition, New York, McGraw-Hill, 2001.

Purpose Convolutionally encode binary data and modulate using PSK method

Library TCM, in Digital Baseband sublibrary of Modulation

Description

The M-PSK TCM Encoder block implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal and mapping the result to a PSK signal constellation.

The **M-ary number** parameter is the number of points in the signal constellation, which also equals the number of possible output symbols from the convolutional encoder. (That is, $\log_2$(**M-ary number**) is equal to n for a rate k/n convolutional code.)

Input Signals and Output Signals

If the convolutional encoder described by the trellis structure represents a rate k/n code, then the block input signal must be a binary column vector with a length of $L*k$ for some positive integer L.

This block accepts a binary-valued input signal. The output signal is a complex column vector of length L.

Specifying the Encoder

To define the convolutional encoder, use the **Trellis structure** parameter. This parameter is a MATLAB structure whose format is described in “Trellis Description of a Convolutional Code” in the Communications System Toolbox documentation. You can use this parameter field in two ways:

- If you want to specify the encoder using its constraint length, generator polynomials, and possibly feedback connection polynomials, then use a `poly2trellis` command within the **Trellis structure** field. For example, to use an encoder with a constraint length of 7, code generator polynomials of 171 and 133 (in octal numbers), and a feedback connection of 171 (in octal), set the **Trellis structure** parameter to

  ```
  poly2trellis(7,[171 133],171)
  ```

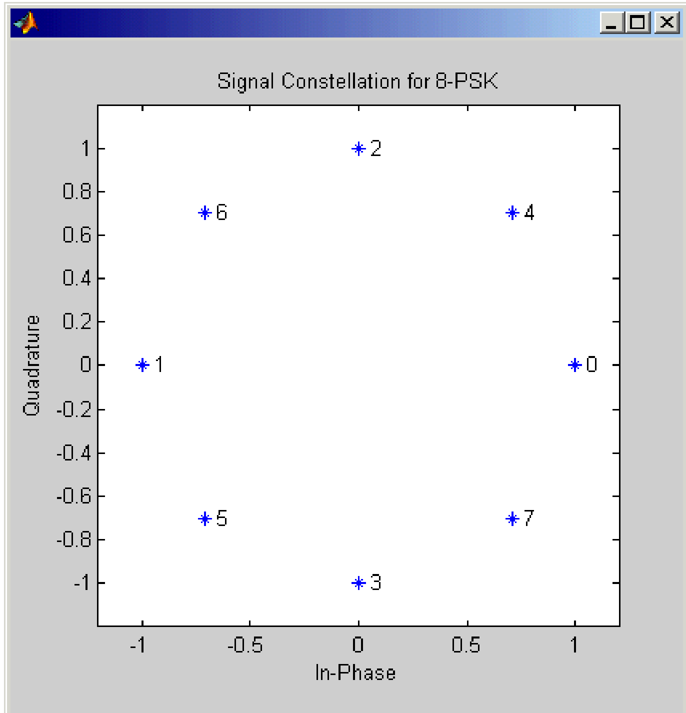
- If you have a variable in the MATLAB workspace that contains the trellis structure, then enter its name as the **Trellis structure** parameter. This way is faster because it causes Simulink software to spend less time updating the diagram at the beginning of each simulation, compared to the usage in the previous bulleted item.

The encoder registers begin in the all-zeros state. You can configure the encoder so that it resets its registers to the all-zeros state during the course of the simulation. To do this, set the `Operation mode` to **Reset on nonzero input via port**. The block then opens a second input port, labeled `Rst`. The signal at the `Rst` port is a scalar signal. When it is nonzero, the encoder resets before processing the data at the first input port.

Signal Constellations

The trellis-coded modulation technique partitions the constellation into subsets called cosets, so as to maximize the minimum distance between pairs of points in each coset. This block internally forms a valid partition based on the value you choose for the **M-ary number** parameter.

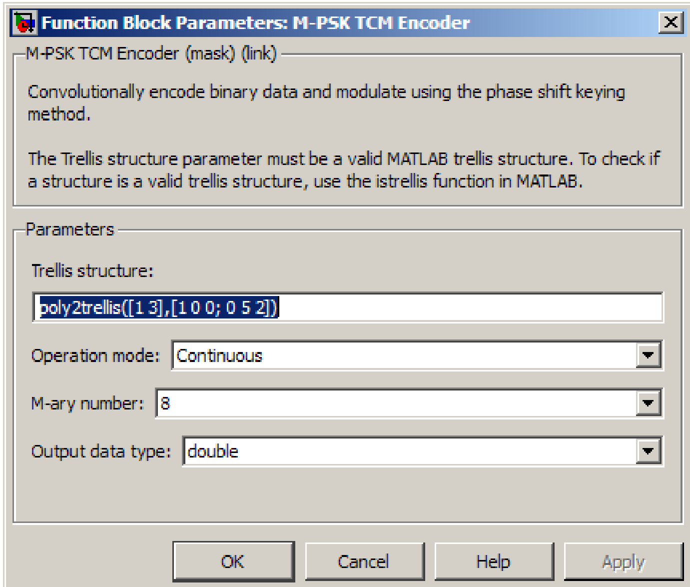
The figure below shows the labeled set-partitioned signal constellation that the block uses when **M-ary number** is 8. For constellations of other sizes, see [1].



Coding Gains

Coding gains of 3 to 6 decibels, relative to the uncoded case can be achieved in the presence of AWGN with multiphase trellis codes [3].

Dialog Box



Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Operation mode

In `Continuous` mode (default setting), the block retains the encoder states at the end of each frame, for use with the next frame.

In `Truncated (reset every frame)` mode, the block treats each frame independently. I.e., the encoder states are reset to all-zeros state at the start of each frame.

In `Terminate trellis by appending bits` mode, the block treats each frame independently. For each input frame, extra bits are used to set the encoder states to all-zeros state at the end of the frame. The output length is given by

$y = n \cdot (x+s)/k$, where x is the number of input bits, and

$s = \text{constraint length} - 1$ (or, in the case of multiple constraint lengths, s =`sum(ConstraintLength(i)-1)`). The block supports this mode for column vector input signals.

In `Reset on nonzero input via port` mode, the block has an additional input port, labeled `Rst`. When the `Rst` input is nonzero, the encoder resets to the all-zeros state.

M-ary number
The number of points in the signal constellation.

Output data type
The output type of the block can be specified as a `single` or `double`. By default, the block sets this to `double`.

Pair Block

M-PSK TCM Decoder

See Also

General TCM Encoder, `poly2trellis`

References

[1] Biglieri, E., D. Divsalar, P. J. McLane and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001

[3] Ungerboeck, G., “Channel Coding with Multilevel/Phase Signals”, *IEEE Trans. on Information Theory*, Vol IT28, Jan. 1982, pp. 55–67.

MSK Demodulator Baseband

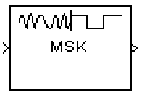
Purpose

Demodulate MSK-modulated data

Library

CPM, in Digital Baseband sublibrary of Modulation

Description



The MSK Demodulator Baseband block demodulates a signal that was modulated using the minimum shift keying method. The input signal is a baseband representation of the modulated signal. The **Phase offset** parameter represents the initial phase of the modulated waveform.

Integer-Valued Signals and Binary-Valued Signals

This block accepts a scalar-valued or column vector input signal with a data type of `single` or `double`. If you set the **Output type** parameter to `Integer`, then the block produces values of 1 and -1. If you set the **Output type** parameter to `Bit`, then the block produces values of 0 and 1.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input width must be an integer multiple of the **Samples per symbol** parameter value, and the input can be a column vector.

- When you set **Output type** to `Bit`, the output width is K times the number of input symbols.
- When you set **Output type** to `Integer`, the output width is the number of input symbols.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is the product of the input sample time and the **Samples per symbol** parameter value.

- When you set **Output type** to `Bit`, the output width equals the number of bits per symbol.

- When you set **Output type** to `Integer`, the output is a scalar.

Traceback Depth and Output Delays

Internally, this block creates a trellis description of the modulation scheme and uses the Viterbi algorithm. The **Traceback depth** parameter, D, in this block is the number of trellis branches used to construct each traceback path. D influences the output delay, which is the number of zero symbols that precede the first meaningful demodulated value in the output.

- When you set the **Rate options** parameter to `Allow multirate processing`, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking`, then the delay consists of D+1 zero symbols.
- When you set the **Rate options** parameter to `Enforce single-rate processing`, then the delay consists of D zero symbols.

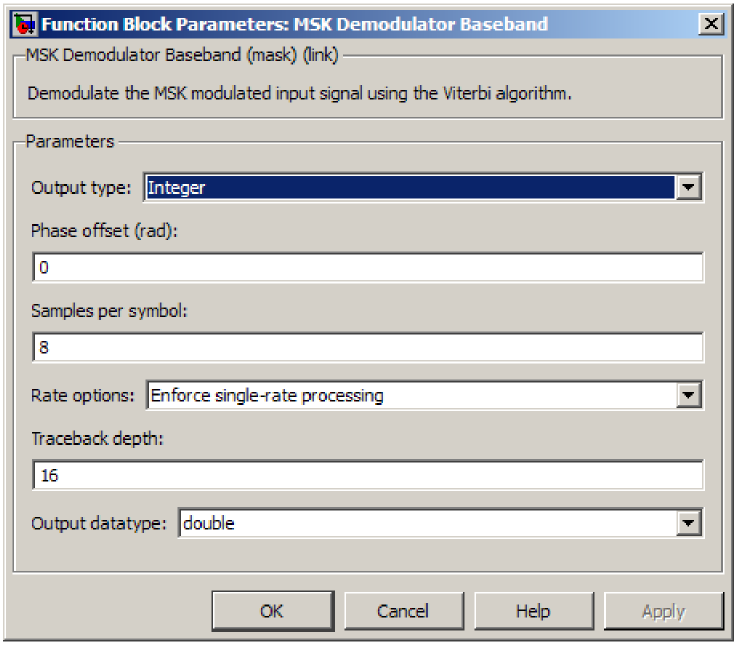
The optimal **Traceback depth** parameter value is dependent on minimum squared Euclidean distance calculations. Alternatively, a typical value, dependent on the number of states, can be chosen using the "five-times-the-constraint-length" rule, which corresponds to $5 \cdot \log 2(numStates)$. The number of states is determined by the following equation:

$$numStates = \begin{Bmatrix} p \cdot 2^{(L-1)}, for\, even\, m \\ 2p \cdot 2^{(L-1)}, for\, odd\, m \end{Bmatrix}$$

where:

- h = m/p is the modulation index proper rational form
  - m = numerator of modulation index
  - p = denominator of modulation index
- L is the Pulse length

Dialog Box



Output type

Determines whether the output consists of bipolar or binary values.

Phase offset (rad)

The initial phase of the modulated waveform.

Samples per symbol

The number of input samples that represent each modulated symbol, which must be a positive integer. For more information, see "Upsample Signals and Rate Changes" in *Communications System Toolbox User's Guide*.

Rate options

Select the rate processing method for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width is the number of symbols (which is given by dividing the input length by the **Samples per symbol** parameter value when the **Output type** parameter is set to `Integer`).

- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output period is the same as the symbol period and equals the product of the input period and the **Samples per symbol** parameter value.

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 *Communications System Toolbox Release Notes* for more information.

For more information, see Single-Rate Processing and Multirate Processing in the Description section of this page.

Traceback depth
The number of trellis branches that the MSK Demodulator Baseband block uses to construct each traceback path.

Output data type
The output data type can be `boolean`, `int8`, `int16`, `int32`, or `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Boolean (When **Output type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (When **Output type** set to `Integer`)

Pair Block

MSK Modulator Baseband

See Also

CPM Demodulator Baseband, Viterbi Decoder

References

[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg, *Digital Phase Modulation*, New York, Plenum Press, 1986.

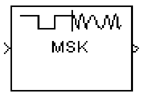
Purpose

Modulate using minimum shift keying method

Library

CPM, in Digital Baseband sublibrary of Modulation

Description



The MSK Modulator Baseband block modulates using the minimum shift keying method. The output is a baseband representation of the modulated signal.

This block accepts a scalar-valued or column vector input signal. For a column vector input signal, the width of the output equals the product of the number of symbols and the value for the **Samples per symbol** parameter.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, then the block accepts values of 1 and -1.

When you set the **Input type** parameter to `Bit`, then the block accepts values of 0 and 1.

For information about the data types each block port supports, see the "Supported Data Types" on page 2-616 table on this page.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. In this mode, the input to the block can be multiple symbols.

- When you set **Input type** to `Integer`, the input can be a column vector, the length of which is the number of input symbols.
- When you set **Input type** to `Bit`, the input width must be an integer multiple of *K*, the number of bits per symbol.

The output width equals the product of the number of input symbols and the **Samples per symbol** parameter value.

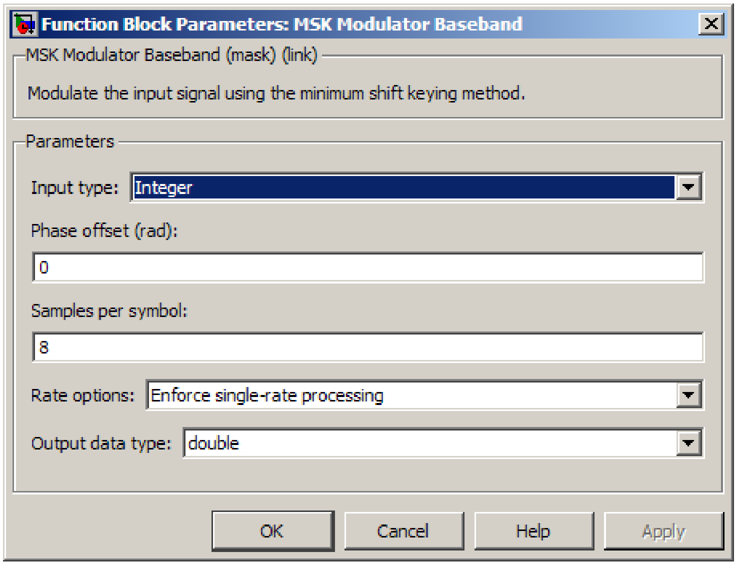
Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

- When you set **Input type** to `Integer`, the input must be a scalar.
- When you set **Input type** to `Bit`, the input width must equal the number of bits per symbol.

The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

Dialog Box



Input type
: Indicates whether the input consists of bipolar or binary values.

Phase offset (rad)
: The initial phase of the output waveform, measured in radians.

Samples per symbol

The number of output samples that the block produces for each integer or binary word in the input, which must be a positive integer. For all non-binary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see “Upsample Signals and Rate Changes” in *Communications System ToolboxUser’s Guide*.

Rate options

Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

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 *Communications System Toolbox Release Notes* for more information.

Output data type

Specify the block output data type as `double` and `single`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Boolean (when **Input type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (when **Input type** set to `Integer`)
Output	• Double-precision floating point • Single-precision floating point

Pair Block

MSK Demodulator Baseband

See Also

CPM Modulator Baseband

References

[1] Anderson, John B., Tor Aulin, and Carl-Erik Sundberg, *Digital Phase Modulation*, New York, Plenum Press, 1986.

Purpose

Recover symbol timing phase using fourth-order nonlinearity method

Library

Timing Phase Recovery sublibrary of Synchronization

Description

The MSK-Type Signal Timing Recovery block recovers the symbol timing phase of the input signal using a fourth-order nonlinearity method. This block implements a general non-data-aided feedback method that is independent of carrier phase recovery but requires prior compensation for the carrier frequency offset. This block is suitable for systems that use baseband minimum shift keying (MSK) modulation or Gaussian minimum shift keying (GMSK) modulation.

Inputs

By default, the block has one input port. The input signal could be (but is not required to be) the output of a receive filter that is matched to the transmitting pulse shape, or the output of a lowpass filter that limits the amount of noise entering this block.

This block accepts a scalar-valued or column vector input signal. The input uses N samples to represent each symbol, where $N > 1$ is the **Samples per symbol** parameter.

- For a column vector input signal, the block operates in single-rate processing mode. In this mode, the output signal inherits its sample rate from the input signal. The input length must be a multiple of N.
- For a scalar input signal, the block operates in multirate processing mode. In this mode, the input and output signals have different sample rates. The output sample rate equals N multiplied by the input sample rate.
- This block accepts input signals of type Double or Single

If you set the **Reset** parameter to `On nonzero input via port`, then the block has a second input port, labeled `Rst`. The `Rst` input determines when the timing estimation process restarts, and must be a scalar.

- If the input signal is a scalar value, the sample time of the `Rst` input equals the symbol period
- If the input signal is a column vector, the sample time of the `Rst` input equals the input port sample time
- This block accepts reset signals of type Double or Boolean

Outputs

The block has two output ports, labeled `Sym` and `Ph`:

- The `Sym` output is the result of applying the estimated phase correction to the input signal. This output is the signal value for each symbol, which can be used for decision purposes. The values in the `Sym` output occur at the symbol rate:
  - For a column vector input signal of length $N*R$, the `Sym` output is a column vector of length R having the same sample rate as the input signal.
  - For a scalar input signal, the sample rate of the `Sym` output equals N multiplied by the input sample rate.
- The `Ph` output gives the phase estimate for each symbol in the input.

  The `Ph` output contains nonnegative real numbers less than N. Noninteger values for the phase estimate correspond to interpolated values that lie between two values of the input signal. The sample time of the `Ph` output is the same as that of the `Sym` output.

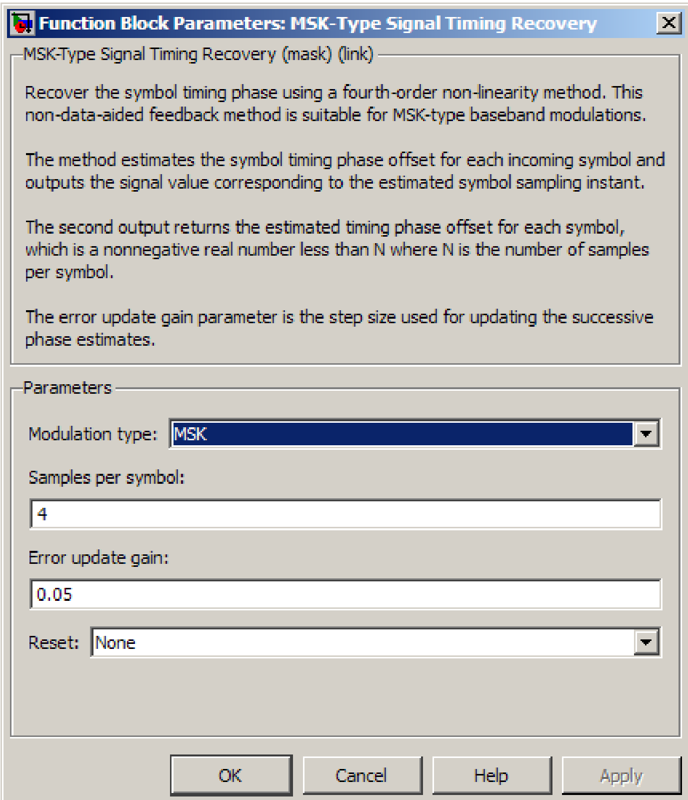
  Note If the `Ph` output is very close to either zero or **Samples per symbol**, or if the actual timing phase offset in your input signal is very close to zero, then the block's accuracy might be compromised by small amounts of noise or jitter. The block works well when the timing phase offset is significant rather than very close to zero.

- The output signal inherits its data type from the input signal.

Delays

When the input signal is a vector, this block incurs a delay of two symbols. When the input signal is a scalar, this block incurs a delay of three symbols.

Dialog Box



Modulation type

The type of modulation in the system. Choices are `MSK` and `GMSK`.

Samples per symbol

The number of samples, N, that represent each symbol in the input signal. This must be greater than 1.

Error update gain

A positive real number representing the step size that the block uses for updating successive phase estimates. Typically, this number is less than 1/N, which corresponds to a slowly varying phase.

This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode. If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. For more information, see Tunable Parameters in the Simulink documentation.

Reset

Determines whether and under what circumstances the block restarts the phase estimation process. Choices are `None`, `Every frame`, and `On nonzero input via port`. The last option causes the block to have a second input port, labeled `Rst`.

Algorithm

This block's algorithm extracts timing information by passing the sampled baseband signal through a fourth-order nonlinearity followed by a digital differentiator whose output is smoothed to yield an error signal. The algorithm then uses the error signal to make the sampling adjustments.

More specifically, this block uses a timing error detector whose result for the kth symbol is e(k), given in [2] by

$$
\begin{aligned}
e(k) = (-1)^{D+1}\,\mathrm{Re}\{r^2(kT - T_s + d_{k-1}) r^{*2}((k-1)T - T_s + d_{k-2})\} \\
-(-1)^{D+1}\,\mathrm{Re}\{r^2(kT + T_s + d_{k-1}) r^{*2}((k-1)T + T_s + d_{k-1})\}
\end{aligned}
$$

$$
\begin{aligned}
e(k) = (-1)^{D+1}\,\mathrm{Re}\{r^2(kT - T_s + d_{k-1}) r^{*2}((k-1)T - T_s + d_{k-2})\} \\
-(-1)^{D+1}\,\mathrm{Re}\{r^2(kT + T_s + d_{k-1}) r^{*2}((k-1)T + T_s + d_{k-1})\}
\end{aligned}
$$

where

- r is the block's input signal
- T is the symbol period
- T_s is the sampling period
- * means complex conjugate
- d_k is the phase estimate for the kth symbol
- D is 1 for MSK and 2 for Gaussian MSK modulation

For more information about the role that e(k) plays in this block's algorithm, see "Feedback Methods for Timing Phase Recovery" in *Communications System Toolbox User's Guide*.

References

[1] D'Andrea, A. N., U. Mengali, and R. Reggiannini, "A Digital Approach to Clock Recovery in Generalized Minimum Shift Keying," *IEEE Transactions on Vehicular Technology*, Vol. 39, No. 3, August 1990, pp. 227-234.

[2] Mengali, Umberto and Aldo N. D'Andrea, *Synchronization Techniques for Digital Receivers*, New York, Plenum Press, 1997.

See Also

Early-Late Gate Timing Recovery, Squaring Timing Recovery

Mueller-Muller Timing Recovery

Purpose

Recover symbol timing phase using Mueller-Muller method

Library

Timing Phase Recovery sublibrary of Synchronization

Description

The Mueller-Muller Timing Recovery block recovers the symbol timing phase of the input signal using the Mueller-Muller method. This block implements a decision-directed, data-aided feedback method that requires prior recovery of the carrier phase.

Inputs

By default, the block has one input port. Typically, the input signal is the output of a receive filter that is matched to the transmitting pulse shape.

This block accepts a scalar-valued or column vector input signal. The input uses N samples to represent each symbol, where $N > 1$ is the **Samples per symbol** parameter.

- For a column vector input signal, the block operates in single-rate processing mode. In this mode, the output signal inherits its sample rate from the input signal. The input length must be a multiple of N.
- For a scalar input signal, the block operates in multirate processing mode. In this mode, the input and output signals have different sample rates. The output sample rate equals N multiplied by the input sample rate.
- This block accepts input signals of type Double or Single

If you set the **Reset** parameter to `On nonzero input via port`, then the block has a second input port, labeled `Rst`. The `Rst` input determines when the timing estimation process restarts, and must be a scalar.

- If the input signal is a scalar value, the sample time of the `Rst` input equals the symbol period
- If the input signal is a column vector, the sample time of the `Rst` input equals the input port sample time

- This block accepts reset signals of type Double or Boolean

Outputs

The block has two output ports, labeled `Sym` and `Ph`:

- The `Sym` output is the result of applying the estimated phase correction to the input signal. This output is the signal value for each symbol, which can be used for decision purposes. The values in the `Sym` output occur at the symbol rate:
  - For a column vector input signal of length $N*R$, the `Sym` output is a column vector of length R having the same sample rate as the input signal.
  - For a scalar input signal, the sample rate of the `Sym` output equals N multiplied by the input sample rate.
- The `Ph` output gives the phase estimate for each symbol in the input.

  The `Ph` output contains nonnegative real numbers less than N. Noninteger values for the phase estimate correspond to interpolated values that lie between two values of the input signal. The sample time of the `Ph` output is the same as that of the `Sym` output.

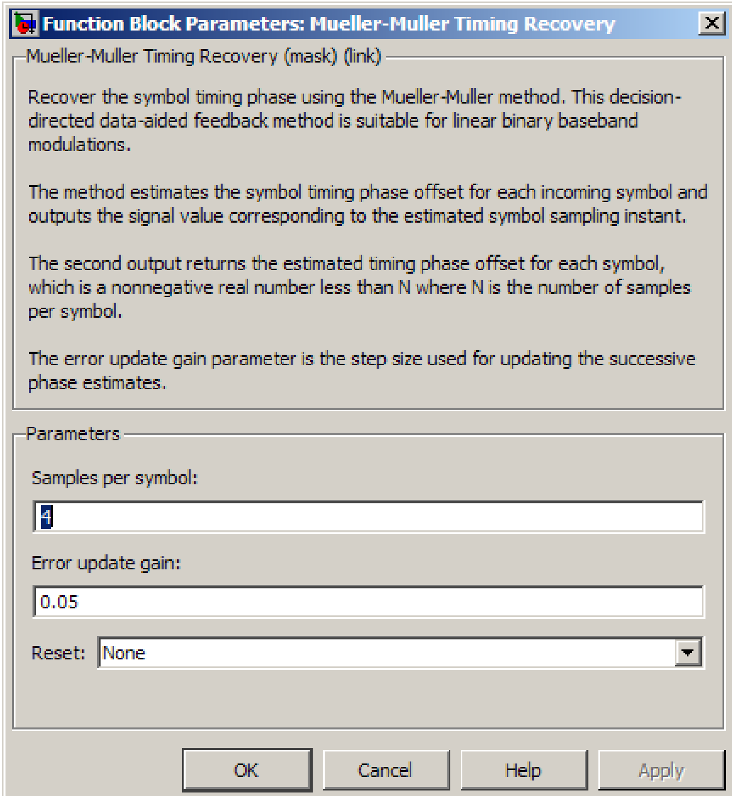
  Note If the `Ph` output is very close to either zero or **Samples per symbol**, or if the actual timing phase offset in your input signal is very close to zero, then the block's accuracy might be compromised by small amounts of noise or jitter. The block works well when the timing phase offset is significant rather than very close to zero.

- The output signal inherits its data type from the input signal.

Delays

When the input signal is a vector, this block incurs a delay of two symbols. When the input signal is a scalar, this block incurs a delay of three symbols.

Dialog Box



Samples per symbol

The number of samples, N, that represent each symbol in the input signal. This must be greater than 1.

Error update gain

A positive real number representing the step size that the block uses for updating successive phase estimates. Typically, this number is less than 1/N, which corresponds to a slowly varying phase.

This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode. If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. For more information, see Tunable Parameters in the Simulink *User's Guide*.

Reset

Determines whether and under what circumstances the block restarts the phase estimation process. Choices are `None`, `Every`, and `On nonzero input via port`. The last option causes the block to have a second input port, labeled `Rst`.

Algorithm

This block uses a timing error detector whose result for the kth symbol is e(k), given by

$$e(k) = \mathrm{Re}\{c_{k-1}^{*} y(kT + d_k) - c_k^{*} y((k-1)T + d_{k-1})\}$$

where

- y is the block's input signal
- c_k is the decision based on the sample value $y(kT+d_k)$
- T is the symbol period
- d_k is the phase estimate for the kth symbol

For more information about the role that e(k) plays in this block's algorithm, see "Feedback Methods for Timing Phase Recovery" in *Communications System ToolboxUser's Guide*.

References

[1] Mengali, Umberto and Aldo N. D'Andrea, *Synchronization Techniques for Digital Receivers*, New York, Plenum Press, 1997.

[2] Meyr, Heinrich, Marc Moeneclaey, and Stefan A. Fechtel, *Digital Communication Receivers*, Vol 2, New York, Wiley, 1998.

[3] Mueller, K. H., and M. S. Muller, "Timing Recovery in Digital Synchronous Data Receivers," *IEEE Transactions on Communications*, Vol. COM-24, May 1976, pp. 516-531.

See Also

Early-Late Gate Timing Recovery, Squaring Timing Recovery

Purpose

Implement μ-law compressor for source coding

Library

Source Coding

Description

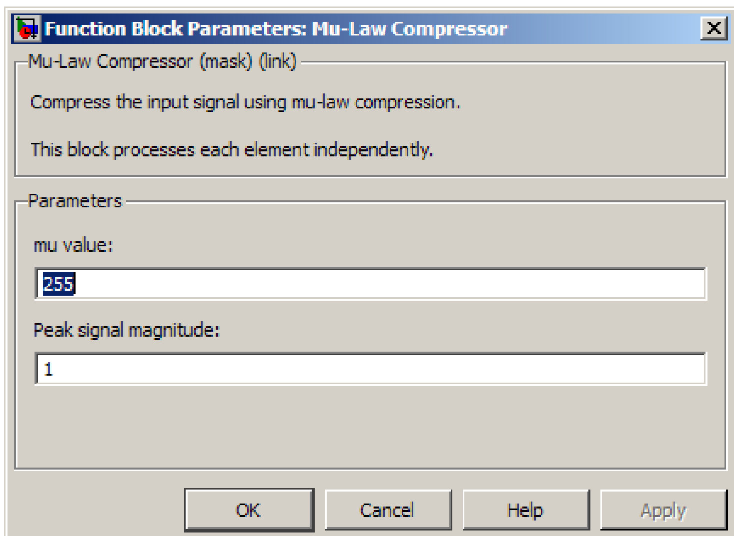
The Mu-Law Compressor block implements a μ-law compressor for the input signal. The formula for the μ-law compressor is

$$y = \frac{V \log(1 + \mu |x| / V)}{\log(1 + \mu)} \operatorname{sgn}(x)$$

where μ is the μ-law parameter of the compressor, V is the peak magnitude of x, log is the natural logarithm, and sgn is the signum function (`sign` in MATLAB).

The input can have any shape or frame status. This block processes each vector element independently.

Dialog Box



mu value

The μ-law parameter of the compressor.

Peak signal magnitude
The peak value of the input signal. This is also the peak value of the output.

Supported Data Type

Port	Supported Data Types
In	• double
Out	• double

Pair Block

Mu-Law Expander

See Also

A-Law Compressor

References

[1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, N.J.: Prentice-Hall, 1988.

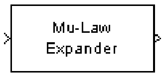
Purpose

Implement μ-law expander for source coding

Library

Source Coding

Description

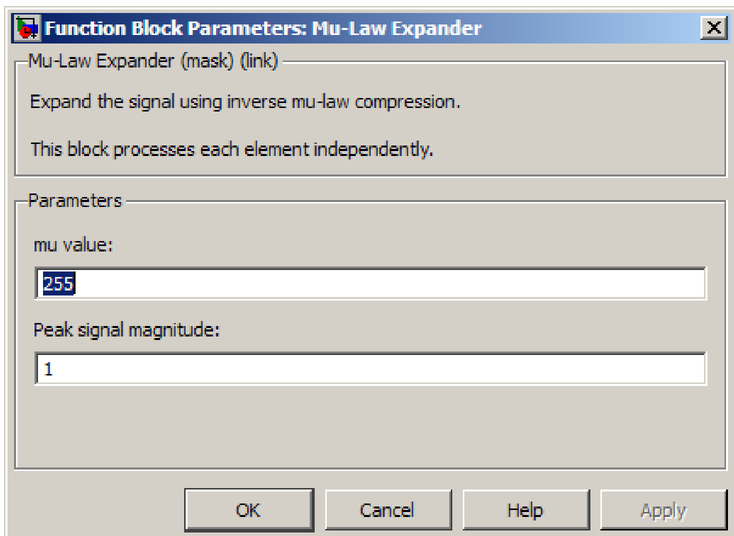


The Mu-Law Expander block recovers data that the Mu-Law Compressor block compressed. The formula for the μ-law expander, shown below, is the inverse of the compressor function.

$$x = \frac{V}{\mu}\left(e^{|y|\log(1+\mu)/V} - 1\right)\mathrm{sgn}(y)$$

The input can have any shape or frame status. This block processes each vector element independently.

Dialog Box



mu value

The μ-law parameter of the compressor.

Peak signal magnitude
The peak value of the input signal. This is also the peak value of the output.

Supported Data Type

Port	Supported Data Types
In	• double
Out	• double

Pair Block

Mu-Law Compressor

See Also

A-Law Expander

References

[1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. Englewood Cliffs, N.J.: Prentice-Hall, 1988.

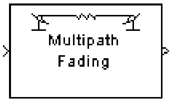
Purpose

Simulate multipath Rayleigh fading propagation channel

Library

Channels

Description



Multipath Rayleigh
Fading Channel

The Multipath Rayleigh Fading Channel block implements a baseband simulation of a multipath Rayleigh fading propagation channel. You can use this block to model mobile wireless communication systems. For details about fading channels, see the references listed below.

This block accepts a scalar value or column vector input signal. The block inherits sample time from the input signal. The input signal must have a discrete sample time greater than 0.

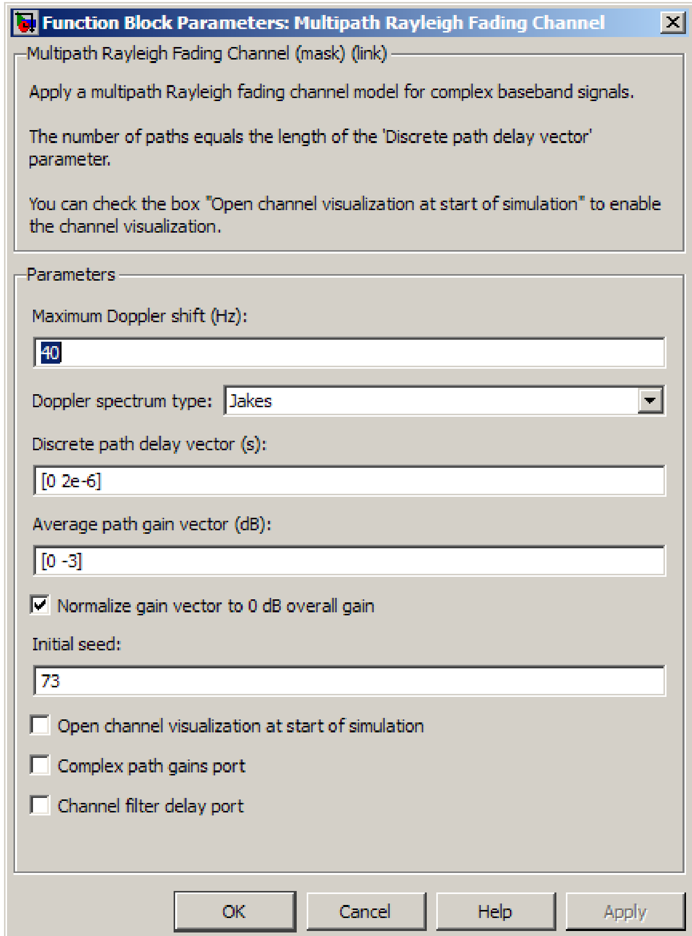
Relative motion between the transmitter and receiver causes Doppler shifts in the signal frequency. You can specify the Doppler spectrum of the Rayleigh process using the **Doppler spectrum type** parameter. For channels with multiple paths, you can assign each path a different Doppler spectrum, by entering a vector of doppler objects in the **Doppler spectrum** field.

Because a multipath channel reflects signals at multiple places, a transmitted signal travels to the receiver along several paths, each of which may have differing lengths and associated time delays. In the block's parameter dialog box, the **Discrete path delay vector** specifies the time delay for each path. If you do not check **Normalize gain vector to 0 dB overall gain**, then the **Average path gain vector** specifies the gain for each path. When you check the box, the block uses a multiple of **Average path gain vector** instead of the **Average path gain vector** itself, choosing the scaling factor so that the channel's effective gain, considering all paths, is 0 dB.

The number of paths indicates the length of **Discrete path delay vector** or **Average path gain vector**, whichever is larger. If both of these parameters are vectors, then they must have the same length; if exactly one of these parameters contains a scalar value, then the block expands it into a vector whose size matches that of the other vector parameter.

The block multiplies the input signal by samples of a Rayleigh-distributed complex random process. The scalar **Initial seed** parameter seeds the random number generator and the block generates random numbers using the Ziggurat method.

Double-clicking this block during simulation with **Inline parameters** off or selecting **Open channel visualization at start of simulation** plots the channel characteristics using the channel visualization tool. See "Channel Visualization" in *Communications System Toolbox User's Guide* for details.



Dialog Box

Maximum Doppler shift (Hz)
A positive scalar value that indicates the maximum Doppler shift.

Doppler spectrum type
Specifies the Doppler spectrum of the Rayleigh process.

This parameter defaults to Jakes Doppler spectrum. Alternatively, you can also choose any of the following types:

- Flat
- Gaussian
- Rounded
- Restricted Jakes
- Asymmetrical Jakes
- Bi-Gaussian
- Bell

For all Doppler spectrum types except `Jakes` and `Flat`, you can choose one or more parameters to control the shape of the spectrum.

You can also select `Specify as dialog parameter` for the **Doppler spectrum type**. Specify the Doppler spectrum by entering an object in the **Doppler spectrum** field. See the `doppler` function reference in *Communications System Toolbox User's Guide* for details on how to construct Doppler objects, and also for the meaning of the parameters associated with the various Doppler spectrum types.

Discrete path delay vector (s)
A vector that specifies the propagation delay for each path.

Average path gain vector (dB)
A vector that specifies the gain for each path.

Normalize gain vector to 0 dB overall gain
Checking this box causes the block to scale the **Gain vector** parameter so that the channel's effective gain (considering all paths) is 0 dB.

Initial seed
The scalar seed for the Gaussian noise generator.

Open channel visualization at start of simulation
Select this check box to open the channel visualization tool when a simulation begins.

Complex path gains port
Select this check box to create a port that outputs the values of the complex path gains for each path. In this N-by-M multichannel output, N represents the number of samples the input signal contains and M represents the number of discrete paths (number of delays).

Channel filter delay port
Select this check box to create a port that outputs the value of the delay (in samples) that results from the filtering operation of this block. This delay is zero if only one path is simulated, but can be greater than zero if more than one path is present. See "Methodology for Simulating Multipath Fading Channels:" in *Communications System Toolbox User's Guide* for a definition of this delay, where it is denoted as N_1.

Algorithm

This implementation is based on the direct-form simulator described in Reference [1]. A detailed explanation of the implementation, including a review of the different Doppler spectra, can be found in [4].

Some wireless applications, such as standard GSM (Global System for Mobile Communication) systems, prefer to specify Doppler shifts in terms of the speed of the mobile. If the mobile moves at speed v making an angle of θ with the direction of wave motion, then the Doppler shift is

$$f_d = (vf/c)\cos\theta$$

where f is the transmission carrier frequency and c is the speed of light. The Doppler frequency represents the maximum Doppler shift arising from motion of the mobile.

Example

Generating Ideal Theoretical BER Results for a Rayleigh Fading Channel

This example illustrates how to generate ideal theoretical BER results for a flat Rayleigh fading channel. The model uses reproduces known theoretical results and shows the correct BER performance for a flat Rayleigh fading channel. In this example, you will run the model and compare the simulation results to the BERTool theoretical results for verification purposes. Note that the EbNo value for the model's AWGN block is 5 dB. You can change the noise power by double-clicking the AWGN block and entering another numeric value in the EbNo parameter.

Opening the Model

You can open the model by clicking here in the MATLAB Help browser. Alternatively, you can type `doc_qpsk_rayleigh_derotated` at the MATLAB command line.

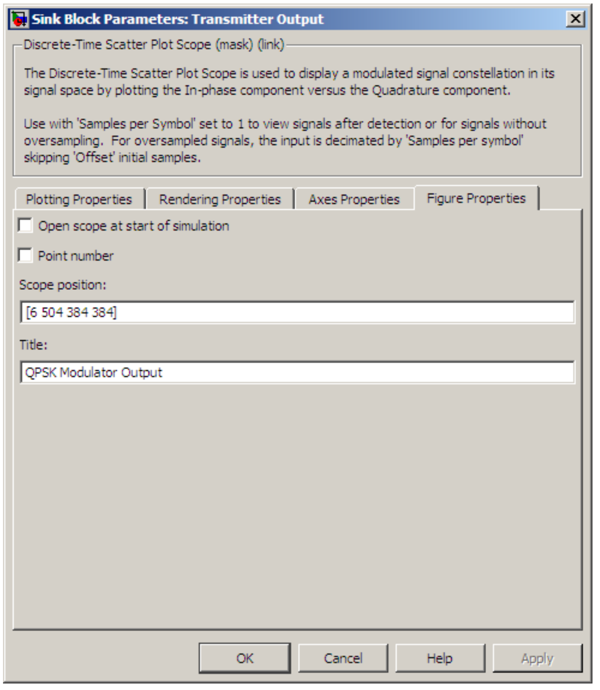
Running the Model and Comparing Results

1. You can run the example by clicking **Simulation > Run**.

2. After the model collects more than 5000 errors, click the stop button.

3. Close the three scopes.

4. In the Simulink model window, double-click the Transmitter Output block. In the mask window, click the **Figure Properties** tab, uncheck **Open scope at start of Simulation**, then click **OK**.



5 In the Simulink model window, double-click the Rayleigh Channel Output block. In the mask window, click the **Figure Properties** tab, uncheck **Open scope at start of Simulation**, then click **OK**.

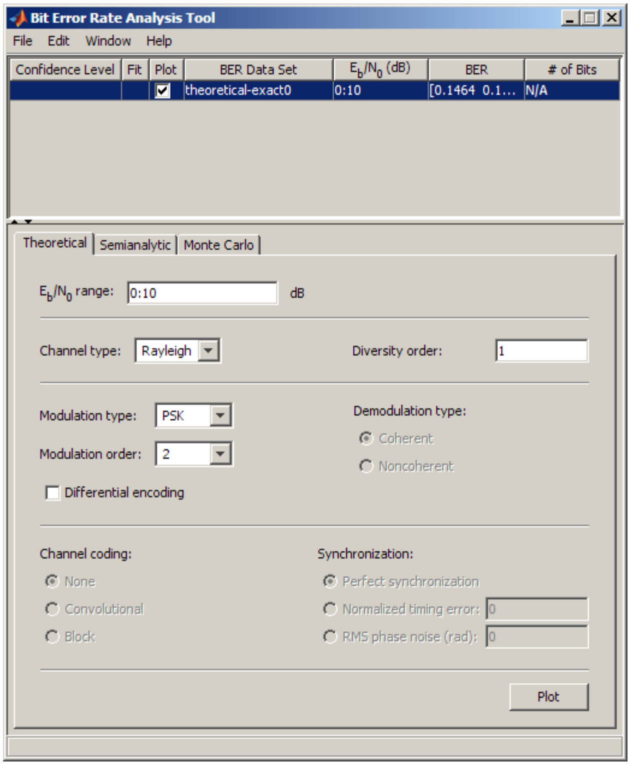
6 In the Simulink model window, double-click the Noisy Rayleigh Channel Output block. In the mask window, click the **Figure Properties** tab, uncheck **Open scope at start of Simulation**, then clock **OK**.

7 In the Simulink model window, double-click the Error Rate Calculation block, check **Stop simulation**, enter 5000 for **Target number of error**, then click **OK**.

8 Click the play button to rerun the example.

9 Open BERTool by typing `bertool` at the MATLAB command line.

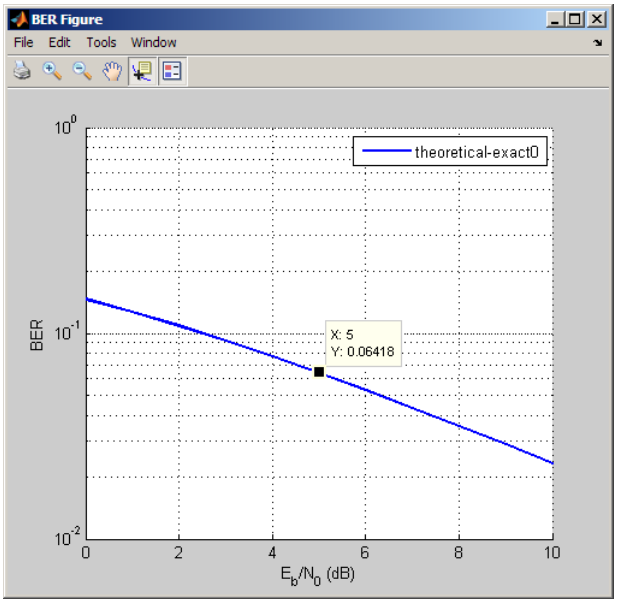
10 In BERTool, click the **Theoretical** tab and make the following selections:



- For **Eb/No range** enter `0:10`
- For **Channel type**, select `Rayleigh`
- For **Diversity Order** enter `1`
- For **Modulation Type**, select `PSK`
- For **Modulation order**, select `4`

11 Click **Plot**.

12 Since the Simulink model uses an EbNo value of 5 dB, verify the probability of error on the BERTool curve at 5 dB. The two values should be approximately equal.



Click the Data Cursor button  (second from right) and click on the BERTool curve at 5dB.

See Also

Rayleigh Noise Generator, Multipath Rician Fading Channel, `doppler`

References

[1] Jeruchim, Michel C., Balaban, Philip, and Shanmugan, K. Sam, *Simulation of Communication Systems*, Second edition, New York, Kluwer Academic/Plenum, 2000.

[2] Jakes, William C., ed. *Microwave Mobile Communications*, New York, IEEE Press, 1974.

[3] Lee, William C. Y., *Mobile Communications Design Fundamentals*, 2nd Ed. New York, Wiley, 1993.

[4] Iskander, Cyril-Daniel, *A MATLAB-based Object-Oriented Approach to Multipath Fading Channel Simulation*, a MATLAB Central submission available from www.mathworks.com.

Purpose

Simulate multipath Rician fading propagation channel

Library

Channels

Description

The Multipath Rician Fading Channel block implements a baseband simulation of a multipath Rician fading propagation channel. You can use this block to model mobile wireless communication systems when the transmitted signal can travel to the receiver along a dominant line-of-sight or direct path. For more details, see "Fading Channels".

This block accepts a scalar value or column vector input signal. The block inherits sample time from the input signal. The input signal must have a discrete sample time greater than 0.

Relative motion between the transmitter and receiver causes Doppler shifts in the signal frequency. You can specify the Doppler spectrum of the Rician process using the **Doppler spectrum type** pop-up menu. For channels with multiple paths, you can assign each path a different Doppler spectrum, by entering a vector of doppler objects in the **Doppler spectrum** field.

Because a multipath channel reflects signals at multiple places, a transmitted signal travels to the receiver along several paths, each of which may have differing lengths and associated time delays. In the block's parameter dialog box, the **Discrete path delay vector** specifies the time delay for each path. If you do not check the **Normalize gain vector to 0 dB overall gain** box, then the **Average path gain vector** specifies the gain for each path. When you check the box, the block uses a multiple of **Average path gain vector** instead of the **Average path gain vector** itself, choosing the scaling factor so that the channel's effective gain considering all paths is 0 dB.

The number of paths indicates the length of **Discrete path delay vector** or **Average path gain vector**, whichever is larger. If both of these parameters are vectors, they must have the same length; if exactly one of these parameters contains a scalar value, the block expands it into a vector whose size matches that of the other vector parameter.

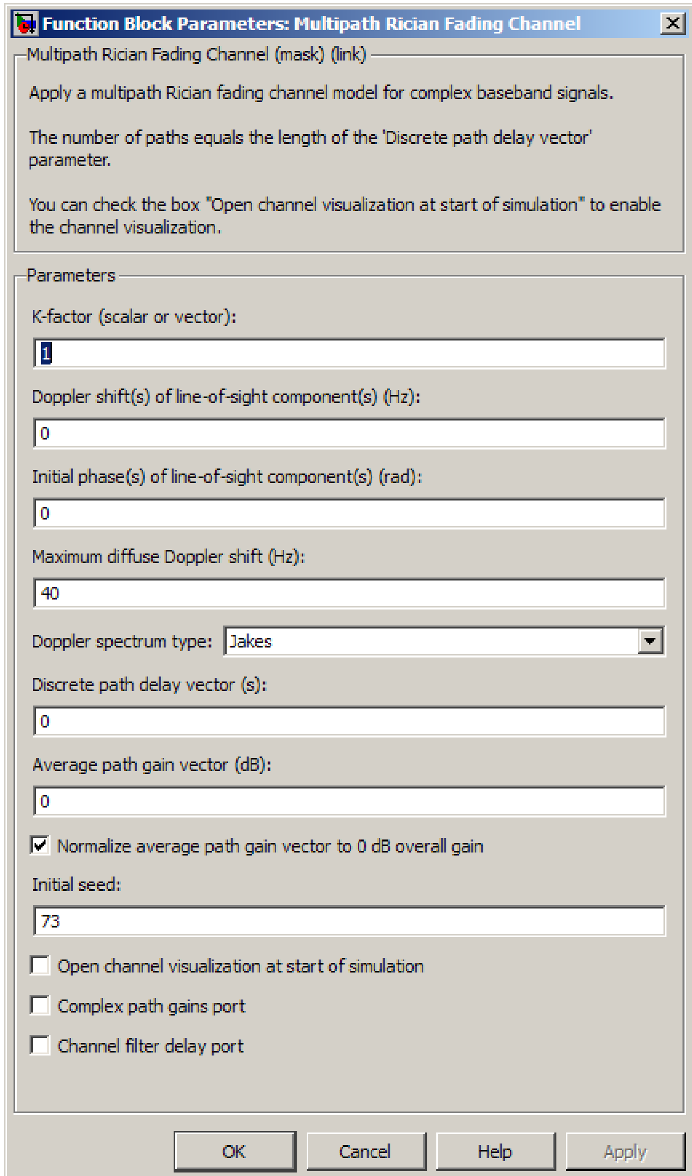
Fading causes the signal to become diffuse. The **K-factor** parameter, which is part of the statistical description of the Rician distribution, represents the ratio between the power in the line-of-sight component and the power in the diffuse component. The ratio is expressed linearly, not in decibels. While the Average path gain vector parameter controls the overall gain through the channel, the **K-factor** parameter controls the gain's partition into line-of-sight and diffuse components.

You can specify the **K-factor** parameter as a scalar or a vector. If the **K-factor** parameter is a scalar, then the first discrete path of the channel is a Rician fading process (it contains a line-of-sight component) with the specified **K-factor**, while the remaining discrete paths indicate independent Rayleigh fading processes (with no line-of-sight component). If the **K-factor** parameter is a vector of the same size as **Discrete path delay vector**, then each discrete path is a Rician fading process with a **K-factor** given by the corresponding element of the vector. You can attribute the line-of-sight component a Doppler shift, through the **Doppler shift(s) of line-of-sight component(s)** parameter, and an initial phase, through the **Initial phase(s) of line-of-sight component(s)**. The **Doppler shift(s) of line-of-sight component(s)** and **Initial phase(s) of line-of-sight component(s)** parameters must be of the same size as the K-factor parameter.

The block multiplies the input signal by samples of a Rician-distributed complex random process. The scalar **Initial seed** parameter seeds the random number generator and the block generates random numbers using the Ziggurat method.

Double-clicking this block during simulation with **Inline parameters** off or selecting the block dialog's check box labeled **Open channel visualization at start of simulation** plots the channel characteristics using the channel visualization tool. See "Channel Visualization" in *Communications System Toolbox User's Guide* for details.

Dialog Box



K-factor

The ratio of power in the line-of-sight component to the power in the diffuse component. The ratio is expressed linearly, not in decibels. If **K-factor** is a scalar value, then the first discrete path is a Rician fading process (it contains a line-of-sight component) with the specified K-factor, while the remaining discrete paths are independent Rayleigh fading processes (with no line-of-sight component). If **K-factor** is a vector of the same size as **Discrete path delay vector**, then each discrete path is a Rician fading process with a **K-factor** given by the corresponding element of the vector.

Doppler shift(s) of line-of-sight components(s) (Hz)

The Doppler shift of the line-of-sight component. It must be a scalar (if **K-factor** is a scalar) or a vector of the same size as **K-factor**. If this parameter contains a scalar value, then the line-of-sight component of the first discrete path has the specified Doppler shift, while the remaining discrete paths become independent Rayleigh fading processes. If the parameter contains a vector, then the line-of-sight component of each discrete path has a Doppler shift given by the corresponding element of the vector.

Initial phase(s) of line-of-sight component(s) (rad)

The initial phase of the line-of-sight component. It must be either a scalar (if **K-factor** is a scalar value) or a vector of the same size as **K-factor**.

Maximum diffuse Doppler shift (Hz)

A positive scalar value that indicates the maximum diffuse Doppler shift.

Doppler spectrum type

Specifies the Doppler spectrum of the Rician process.

This parameter defaults to `Jakes` Doppler spectrum. Alternately, you can choose any of the following types:

- Flat
- Gaussian
- Rounded
- Restricted Jakes

- Asymmetrical Jakes
- Bi-Gaussian
- Bell

For all Doppler spectrum types except `Jakes` and `Flat`, You can use one or more parameters to control the shape of the spectrum.

You can also select `Specify as dialog parameter` for the **Doppler spectrum type**. Specify the Doppler spectrum by entering an object in the **Doppler spectrum** field. See the `doppler` function reference in *Communications System Toolbox User's Guide* for details on how to construct doppler objects, and for the meaning of the parameters associated with the various Doppler spectrum types.

Discrete delay vector(s)
A vector that specifies the propagation delay for each path.

Average path gain vector (dB)
A vector that specifies the gain for each path.

Initial seed
The scalar seed for the Gaussian noise generator.

Open channel visualization at start of simulation
Select this check box to open the channel visualization tool when a simulation begins. This block supports channel visualization for a column vector input signal.

Complex path gains port
Select this check box to create a port that outputs the values of the complex path gains for each path. In this N-by-M multichannel output, N represents the number of samples the input contains and M represents the number of discrete paths (number of delays).

Channel filter delay port
Select this check box to create a port that outputs the value of the delay (in samples) that results from the filtering operation of this block. This delay is zero if only one path is simulated,

but can be greater than zero if more than one path is present. See "Methodology for Simulating Multipath Fading Channels:" in *Communications System Toolbox User's Guide* for a definition of this delay, where it is denoted as N_1.

Algorithm

This implementation is based on the direct form simulator described in Reference [1]. A detailed explanation of the implementation, including a review of the different Doppler spectra, can be found in [4].

Some wireless applications, such as standard GSM (Global System for Mobile Communication) systems, prefer to specify Doppler shifts in terms of the speed of the mobile. If the mobile moves at speed v making an angle of θ with the direction of wave motion, the Doppler shift is

$$f_{\mathrm{d}} = (vf/c)\cos\theta$$

where f is the transmission carrier frequency and c is the speed of light. The Doppler frequency is the maximum Doppler shift arising from the motion of the mobile.

See Also

Rician Noise Generator, Multipath Rayleigh Fading Channel, `doppler`

References

[1] Jeruchim, Michel C., Balaban, P., and Shanmugan, K. Sam, *Simulation of Communication Systems*, Second edition, New York, Kluwer Academic/Plenum, 2000.

[2] Jakes, William C., ed., *Microwave Mobile Communications*, New York, IEEE Press, 1974.

[3] Lee, William C. Y., *Mobile Communications Design Fundamentals*, 2nd ed., New York, John Wiley & Sons, Inc., 1993.

[4] Iskander, Cyril-Daniel, *A MATLAB-based Object-Oriented Approach to Multipath Fading Channel Simulation*, a MATLAB Central submission available from www.mathworks.com.

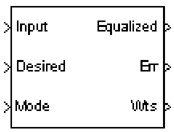
Purpose

Equalize using decision feedback equalizer that updates weights with normalized LMS algorithm

Library

Equalizer Block

Description



The Normalized LMS Decision Feedback Equalizer block uses a decision feedback equalizer and the normalized LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the normalized LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced (i.e. T-spaced) equalizer. When you set the **Number of samples per symbol** parameter to a value greater than 1, , the weights are updated once every N^{th} sample, for a T/N-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of forward taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input, as described in “Reference Signal and Operation Modes” in *Communications System Toolbox User’s Guide*.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output, as described in “Adaptive Algorithms” in *Communications System Toolbox User’s Guide*.

Decision-Directed Mode and Training Mode

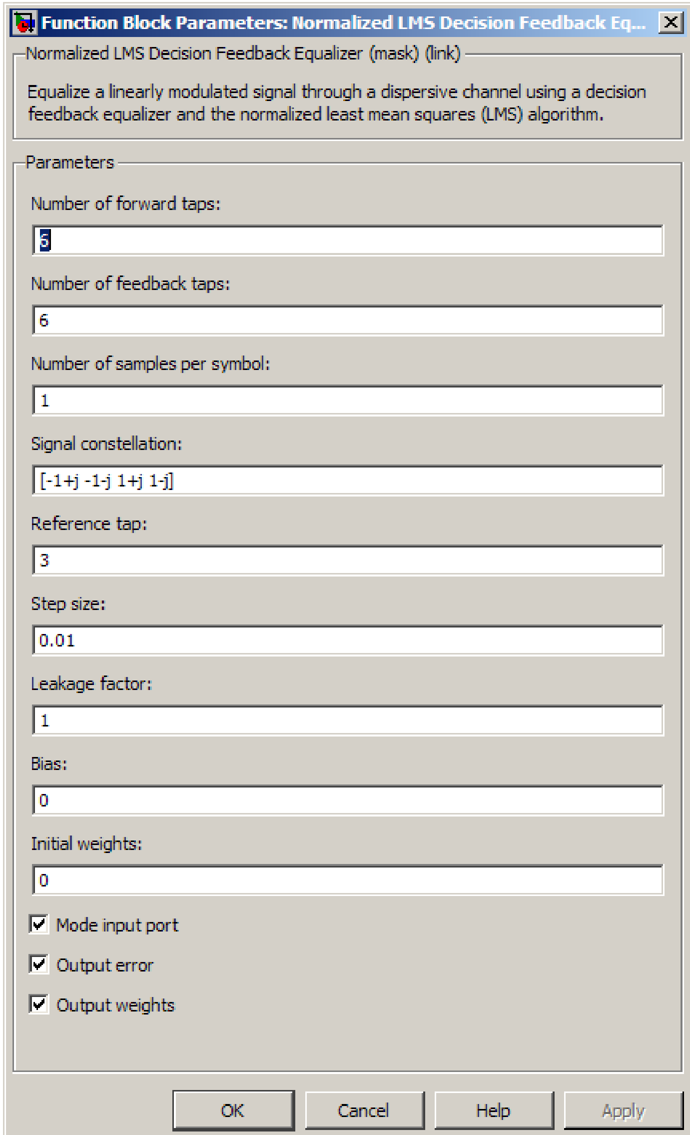
To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Using Adaptive Equalizers" in *Communications System Toolbox User's Guide*.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap of the forward filter.



Dialog Box

Number of forward taps

The number of taps in the forward filter of the decision feedback equalizer.

Number of feedback taps
The number of taps in the feedback filter of the decision feedback equalizer.

Number of samples per symbol
The number of input samples for each symbol.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulation.

Reference tap
A positive integer less than or equal to the number of forward taps in the equalizer.

Step size
The step size of the normalized LMS algorithm.

Leakage factor
The leakage factor of the normalized LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Bias
The bias parameter of the normalized LMS algorithm, a nonnegative real number. This parameter is used to overcome difficulties when the algorithm's input signal is small.

Initial weights
A vector that concatenates the initial weights for the forward and feedback taps.

Mode input port
If you select this check box, the block has an input port that enables you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. The equalizer will train for the length of the Desired signal. If the mode input is not present, the equalizer will train at the beginning of every frame for the length of the Desired signal.

Output error

If you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

If you select this check box, the block outputs the current forward and feedback weights, concatenated into one vector.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

See Also

Normalized LMS Linear Equalizer, LMS Decision Feedback Equalizer

Normalized LMS Linear Equalizer

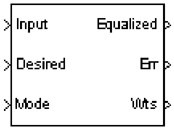
Purpose

Equalize using linear equalizer that updates weights with normalized LMS algorithm

Library

Equalizers

Description



The Normalized LMS Linear Equalizer block uses a linear equalizer and the normalized LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the normalized LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, the block implements a symbol-spaced (i.e. T-spaced) equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than `1`, the weights are updated once every N^{th} sample, for a T/N-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input, as described in "Reference Signal and Operation Modes" in *Communications System Toolbox User's Guide*.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.

- `Weights` output, as described in "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Decision-Directed Mode and Training Mode

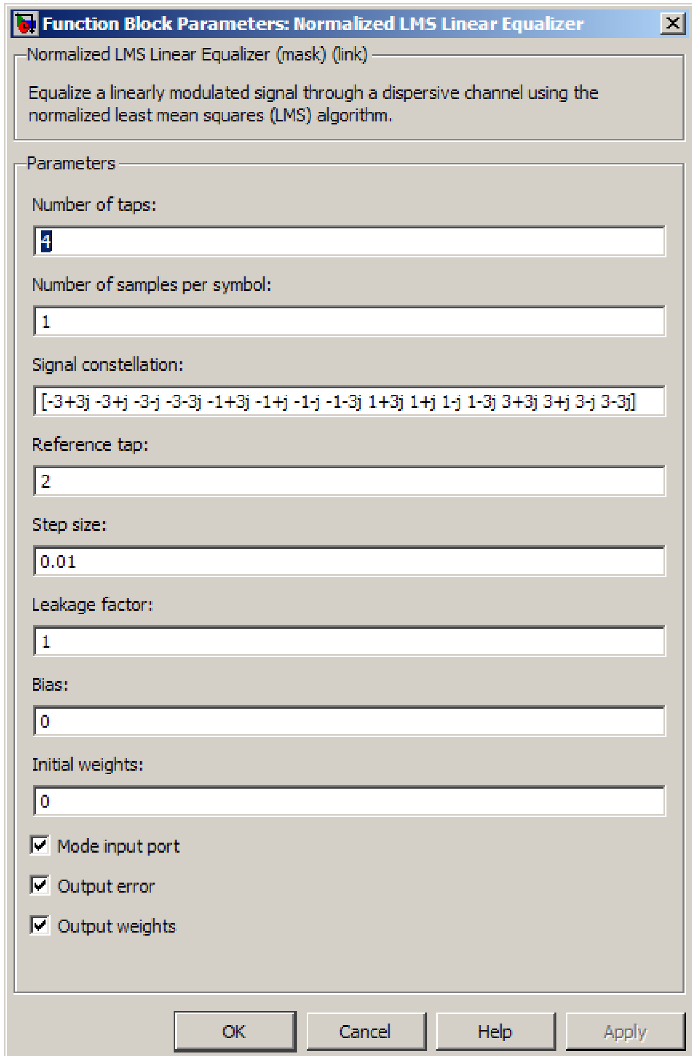
To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Using Adaptive Equalizers" in *Communications System Toolbox User's Guide*.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap.



Dialog Box

Number of taps

The number of taps in the filter of the linear equalizer.

Number of samples per symbol
The number of input samples for each symbol.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulation.

Reference tap
A positive integer less than or equal to the number of taps in the equalizer.

Step size
The step size of the normalized LMS algorithm.

Leakage factor
The leakage factor of the normalized LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Bias
The bias parameter of the normalized LMS algorithm, a nonnegative real number. This parameter is used to overcome difficulties when the algorithm's input signal is small.

Initial weights
A vector that lists the initial weights for the taps.

Mode input port
When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error
If you check this box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

If you check this box, the block outputs the current weights.

Examples

See the Adaptive Equalization example.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

See Also

Normalized LMS Decision Feedback Equalizer, LMS Linear Equalizer

Purpose

Demodulate OQPSK-modulated data

Library

PM, in Digital Baseband sublibrary of Modulation

Description

The OQPSK Demodulator Baseband block demodulates a signal that was modulated using the offset quadrature phase shift keying method. The input is a baseband representation of the modulated signal.

The input must be a discrete-time complex signal. This block accepts a scalar-valued or column vector input signal. For information about the data types each block port supports, see "Supported Data Types" on page 2-666.

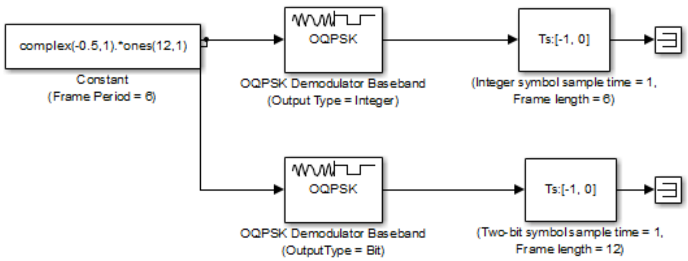
When you set the **Output type** parameter to `Integer`, the block outputs integer symbol values between 0 and 3. When you set the **Output type** parameter to `Bit`, the block outputs a 2–bit binary representation of integers, in a binary-valued vector with a length that is an even number.

The block produces one output symbol for each pair of input samples. The input sample period is half the period of each output integer or bit pair. The constellation used to map bit pairs to symbols is on the reference page for the OQPSK Modulator Baseband block.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input can be an even length column vector.

- When you set **Output type** to `Bit`, the output width is 2 times the number of input symbols.
- When you set **Output type** to `Integer`, the output width is the number of input symbols.

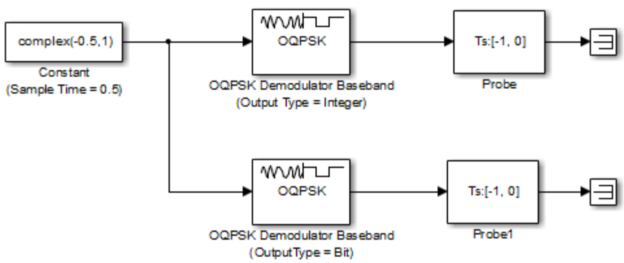


To open this model, type `doc_moqpskdemod_fb` at the MATLAB command line.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is two times the input sample time.

- When you set **Output type** to `Bit`, the output width equals 2.
- When you set **Output type** to `Integer`, the output is a scalar.

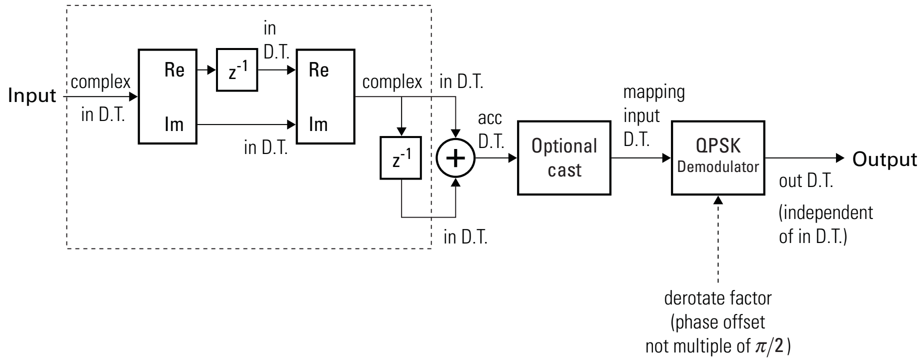


To open this model, type doc_moqpskdemod_sb at the MATLAB command line.

Delays

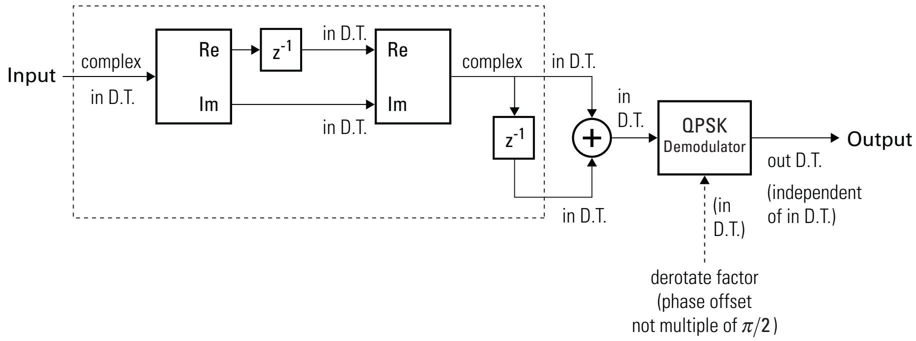
The modulator-demodulator pair incurs a delay, as described in "Example: Delays from Demodulation".

Signal Flow Diagram



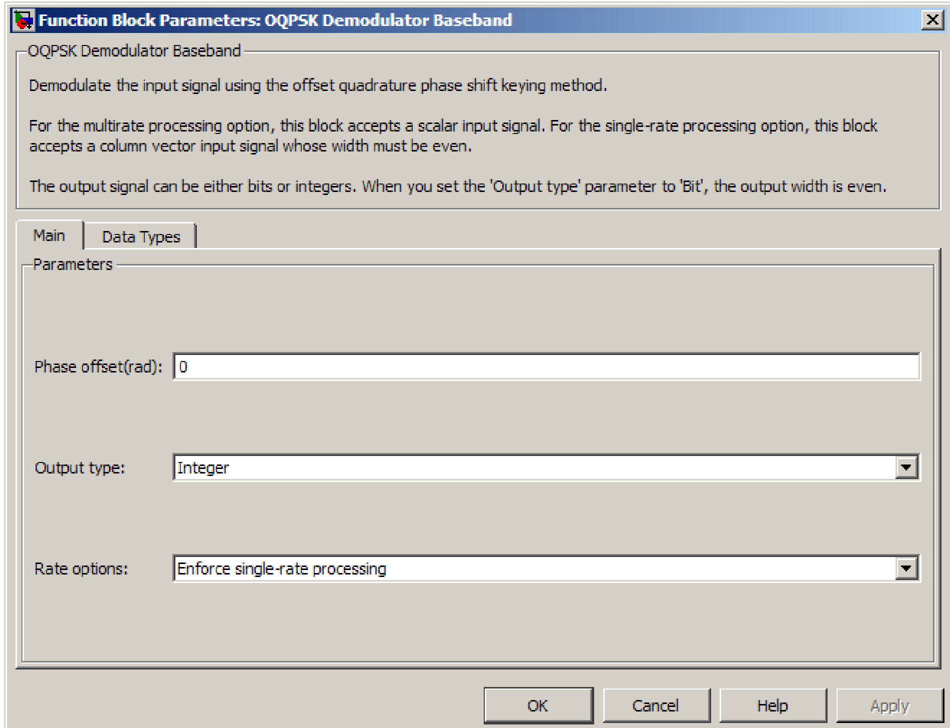
OQPSK Fixed-Point Signal Flow Diagram

Note Every two input samples produce one output symbol. In the preceding figure, the dotted line represents the region comprised of input sample processing.



OQPSK Floating Point Signal Flow Diagram

Note Every two input samples produce one output symbol. In the preceding figure, the dotted line represents the region comprised of input sample processing.



Dialog Box

Phase offset (rad)

The amount by which the phase of the zeroth point of the signal constellation is shifted from π/4.

Output type

Determines whether the output consists of integers or pairs of bits.

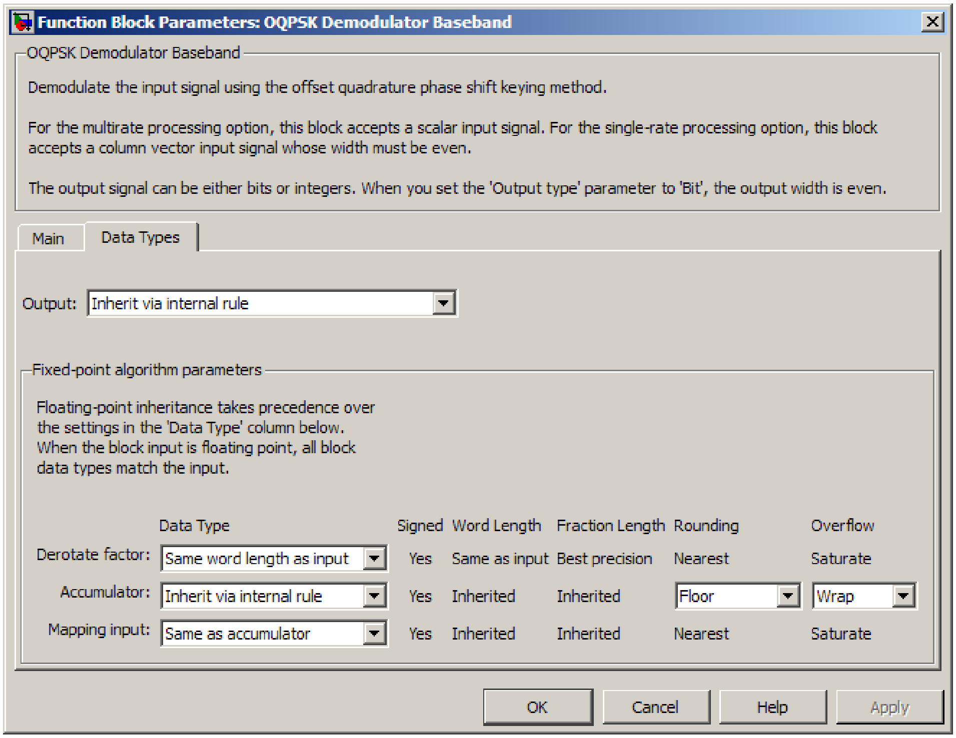
Rate options

Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals half the input width for integer outputs.

- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output symbol time is two times the input sample time.

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 *Communications System Toolbox Release Notes* for more information.



Data Type Pane

Output

For bit outputs, the output data type can be set to `'Inherit via internal rule'`, `'Smallest unsigned integer'`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `boolean`.

For integer outputs, the output data type can be set to `'Inherit via internal rule'`, `'Smallest unsigned integer'`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

When this parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is a floating-point type (single or double). If the input data type is fixed-point, the output data type will work as if this parameter is set to 'Smallest unsigned integer'.

When this parameter is set to 'Smallest unsigned integer', the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model.

If ASIC/FPGA is selected in the **Hardware Implementation** pane, and **Output type** is Bit, the output data type is the ideal minimum one-bit size, i.e., ufix1. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding to the size of a char (e.g., uint8).

If ASIC/FPGA is selected in the **Hardware Implementation** pane, and **Output type** is Integer, the output data type is the ideal minimum two-bit size, i.e., ufix2. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit two bits, usually corresponding to the size of a char (e.g., uint8).

Derotate factor

This parameter only applies when the input is fixed-point and **Phase offset** is not a multiple of $\frac{\Pi}{2}$.

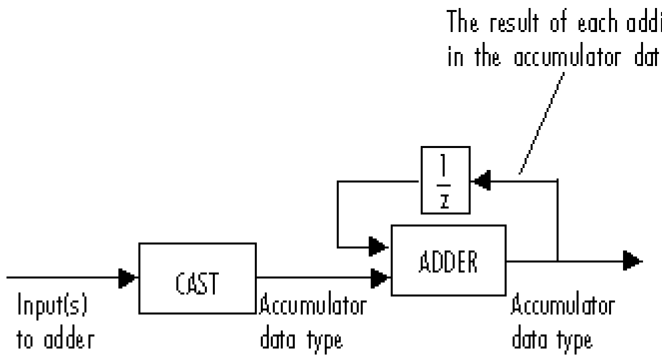
This can be set to Same word length as input or Specify word length, in which case a field is enabled for user input.

Accumulator

Specify the data type for the Accumulator. You can set this parameter to Inherit via internal rule, Same as input or Binary point scaling.

The **Accumulator** parameter only applies for fixed-point inputs. The selections you make for the **Rounding** and **Overflow** parameters affect the **Accumulator**.

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



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

- When you select `Inherit via internal rule`, the accumulator output word and fraction lengths are automatically calculated for you. Refer to "Inherit via Internal Rule" for more information.
- 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 first input to the block.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

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

Rounding
The block uses the **Rounding** method when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. For more information, see “Rounding Modes” in DSP System Toolbox User’s Guide or “Rounding Mode: Simplest” in the Fixed-Point Designer documentation.

Overflow
Specify the method of storing the result when the magnitude of a fixed-point calculation result that does not does not fit within the range of the data type selected. You can select either `Wrap` or `Saturate` for this parameter.

For more information refer to Overflow in the Precision and Range subsection of DSP System Toolbox.

Mapping input
This can be set to `Same as accumulator` or `Binary point scaling`. This parameter only applies for fixed-point inputs.

Supported Data Types

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

Port	Supported Data Types
	• Boolean when **Output type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) in ASIC/FPGA when **Output type** is `Bit` • ufix(2) in ASIC/FPGA when **Output type** is `Integer`

Pair Block

OQPSK Modulator Baseband

See Also

QPSK Demodulator Baseband

OQPSK Modulator Baseband

Purpose

Modulate using offset quadrature phase shift keying method

Library

PM, in Digital Baseband sublibrary of Modulation

Description

The OQPSK Modulator Baseband block modulates using the offset quadrature phase shift keying method. The block outputs a baseband representation of the modulated signal.

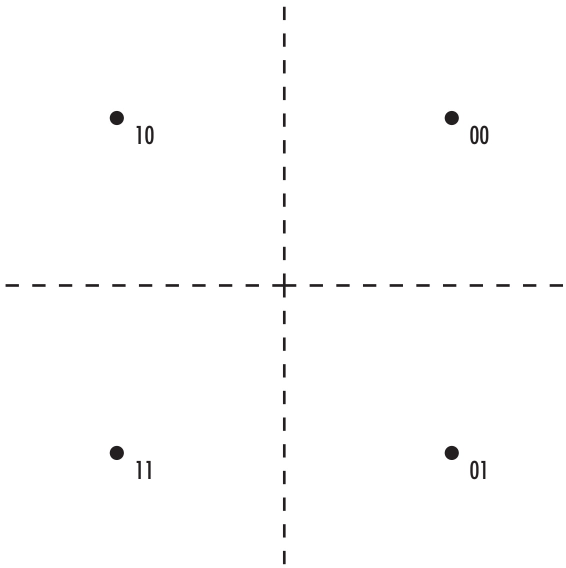
Note The OQPSK modulator block upsamples by a factor of 2.

When you set the **Input type** parameter to `Integer`, valid input values are `0`, `1`, `2`, and `3`. In this case, the block accepts a scalar or a column vector input signal.

When you set the **Input type** parameter to `Bit`, a binary-valued vector is a valid input value. In this case, the block accepts a column vector input signal with a length that is an even integer.

For information about the data types each block port supports, see "Supported Data Types" on page 2-674.

The constellation the block uses to map bit pairs to symbols is shown in the following figure. If you set the **Phase offset** parameter to a nonzero value, then the constellation rotates by that value.



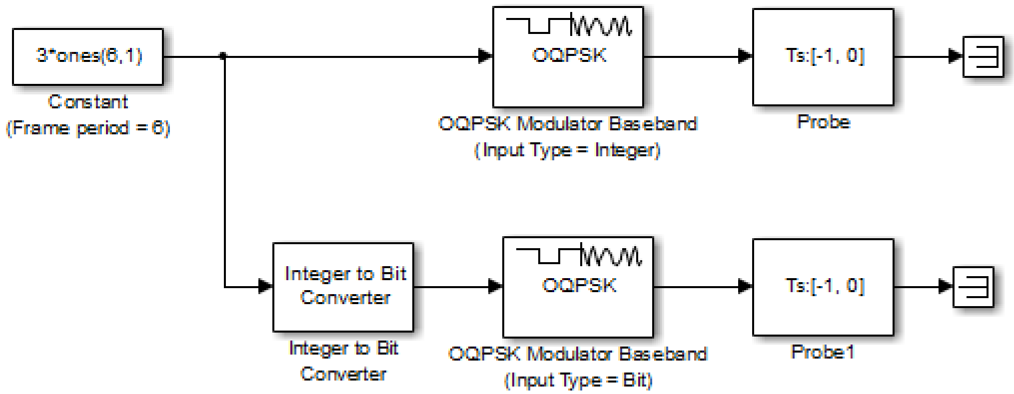
Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. In this mode, the input to the block can be multiple symbols.

- When you set **Input type** to `Integer`, the input can be a scalar value or column vector, the length of which is the number of input symbols.
- When you set **Input type** to `Bit`, the input width must be an integer multiple of two.

The output sample period is half the period of each integer or bit pair in the input.

OQPSK Modulator Baseband



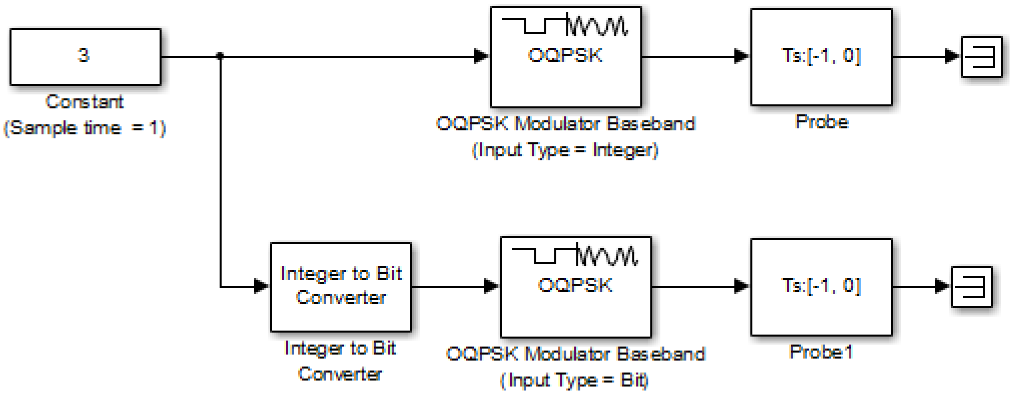
To open this model, type `doc_moqpskmod_fb` at the MATLAB command line.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

- When you set **Input type** to `Integer`, the input must be a scalar value.
- When you set **Input type** to `Bit`, the input width must equal 2.

The output sample time equals one-half the symbol period. The first output symbol is an initial condition of zero that is unrelated to the input values.

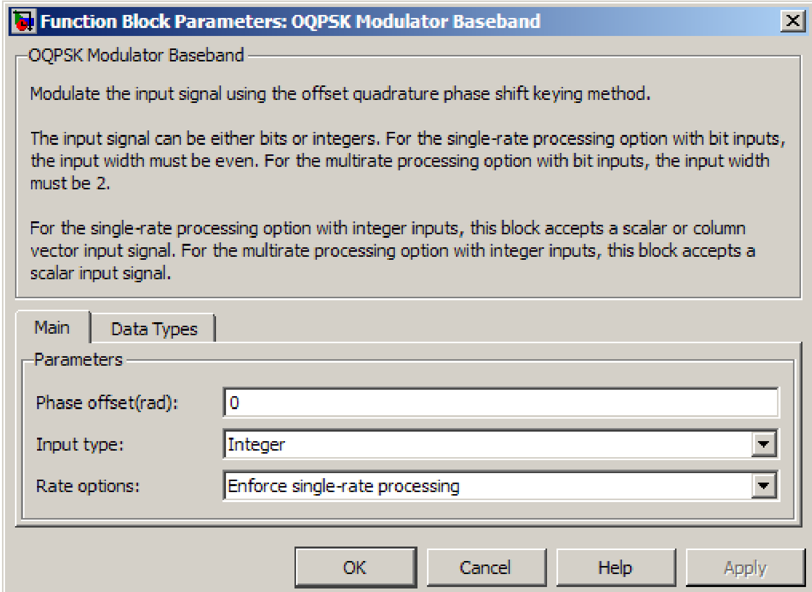


To open this model, type `doc_moqpskmod_sb` at the MATLAB command line.

Delays

The modulator-demodulator pair incurs a delay, as described in “Delays in Digital Modulation”.

Dialog Box



Phase offset (rad)

The amount by which the block shifts the phase of the zeroth point of the signal constellation from π/4.

Input type

Indicates whether the input consists of integers or pairs of bits.

Rate options

Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals two times the number of symbols for integer inputs.

- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals one-half the symbol period.

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 *Communications System Toolbox Release Notes* for more information.

Output data type
Select the output data type as `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

Setting this parameter to `Fixed-point` or `User-defined` enables fields in which you can further specify details. Setting this parameter to `Inherit via back propagation`, sets the output data type and scaling to match the following block.

Output 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 signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set output fraction length to
Specify the scaling of the fixed-point output by either of the following methods:

- When you select `Best precision` the block sets the output scaling so the output signal has the best possible precision.

- When you select `User-defined` you specify the output scaling using the **Output fraction length** parameter.

This parameter applies when you select `Fixed-point` for the **Output data type** parameter or you select `User-defined` and the specified output data type is a fixed-point data type.

Output 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 applies when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set output fraction length to** parameter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean when **Input type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) when **Input type** is `Bit` • ufix(2) when **Input type** is `Integer`
Output	• Double-precision floating point • Single-precision floating point • Signed fixed point

Pair Block

OQPSK Demodulator Baseband

See Also

QPSK Modulator Baseband

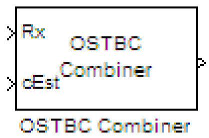
Purpose

Combine inputs for received signals and channel estimate according to orthogonal space-time block code (OSTBC)

Library

MIMO

Description

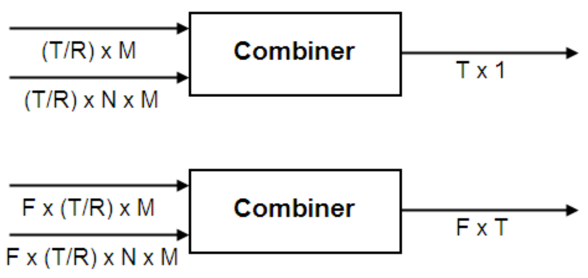


The OSTBC Combiner block combines the input signal (from all of the receive antennas) and the channel estimate signal to extract the soft information of the symbols that were encoded using an OSTBC. The input channel estimate may not be constant during each codeword block transmission and the combining algorithm uses only the estimate for the first symbol period per codeword block. A symbol demodulator or decoder would follow the Combiner block in a MIMO communications system.

The block conducts the combining operation for each symbol independently. The combining algorithm depends on the structure of the OSTBC. For more information, see the OSTBC Combining Algorithms section of this help page.

Dimension

Along with the time and spatial domains for OSTBC transmission, the block supports an optional dimension, over which the combining calculation is independent. This dimension can be thought of as the frequency domain for OFDM-based applications. The following illustration indicates the supported dimensions for inputs and output of the OSTBC Combiner block.



The following table describes each variable for the block.

Variable	Description
F	The additional dimension; typically the frequency dimension. The combining calculation is independent of this dimension.
N	Number of transmit antennas.
M	Number of receive antennas.
T	Output symbol sequence length in time domain.
R	Symbol rate of the code.

Note On the two inputs, T/R is the symbol sequence length in the time domain.

F can be any positive integers. M can be 1 through 8, indicated by the **Number of receive antennas** parameter. N can be 2, 3 or 4, indicated by the **Number of transmit antennas** parameter. The time domain length T/R must be a multiple of the codeword block length (2 for Alamouti; 4 for all other OSTBC). For $N = 2$, T/R must be a multiple of 2. When $N > 2$, T/R must be a multiple of 4. R defaults to 1 for 2 antennas. R can be either $\frac{3}{4}$ or $\frac{1}{2}$ for more than 2 antennas.

The supported dimensions for the block depend upon the values of F and M. For one receive antenna ($M = 1$), the received signal input must be a column vector or a full 2–D matrix, depending on the value for F. The corresponding channel estimate input must be a full 2-D or 3-D matrix.

For more than one receive antenna ($M > 1$), the received signal input must be a full 2-D or 3-D matrix, depending on the value for F. Correspondingly, the channel estimate input must be a 3-D or 4-D matrix, depending on the value for F.

To understand the block's dimension propagation, refer to the following table.

	Input 1 (Received Signal)	Input 2 (Channel Estimate)	Output
F = 1 and M = 1	Column vector	2-D	Column vector
F = 1 and M > 1	2-D	3-D	Column vector
F > 1 and M = 1	2-D	3-D	2-D
F > 1 and M > 1	3-D	4-D	2-D

Data Type

For information about the data types each block port supports, see the "Supported Data Type" on page 2-686 table on this page. The output signal inherits the data type from the inputs. The block supports different fixed-point properties for the two inputs. For fixed-point signals, the output word length and fractional length depend on the block's mask parameter settings. See Fixed-Point Signals for more information about fixed-point data propagation of this block.

Frames

The output inherits the frameness of the received signal input. For either column vector or full 2-D matrix input signal, the input can be either frame-based or sample-based. A 3–D or 4–D matrix input signal must have sample-based input.

OSTBC Combining Algorithms

The OSTBC Combiner block supports five different OSTBC combining computation algorithms. Depending on the selection for **Rate** and **Number of transmit antennas**, you can select one of the algorithms shown in the following table.

Transmit Antenna	Rate	Computational Algorithm per Codeword Block Length
2	1	$\begin{pmatrix} \hat{s}_1 \\ \hat{s}_2 \end{pmatrix} = \frac{1}{\lVert H \rVert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} \end{pmatrix}.$
3	1/2	$\begin{pmatrix} \hat{s}_1 \\ \hat{s}_2 \end{pmatrix} = \frac{1}{\lVert H \rVert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} + h^*_{3,j} r_{3,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} - h_{3,j} r^*_{4,j} \end{pmatrix}.$
3	3/4	$\begin{pmatrix} \hat{s}_1 \\ \hat{s}_2 \\ \hat{s}_3 \end{pmatrix} = \frac{1}{\lVert H \rVert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} - h_{3,j} r^*_{3,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} - h_{3,j} r^*_{4,j} \\ h^*_{3,j} r_{1,j} + h_{1,j} r^*_{3,j} + h_{2,j} r^*_{4,j} \end{pmatrix}.$
4	1/2	$\begin{pmatrix} \hat{s}_1 \\ \hat{s}_2 \end{pmatrix} = \frac{1}{\lVert H \rVert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} + h^*_{3,j3,j} r_{3,j} + h_{4,j} r^*_{4,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} + h^*_{4,j} r_{3,j} - h_{3,j} r^*_{4,j} \end{pmatrix}.$
4	3/4	$\begin{pmatrix} \hat{s}_1 \\ \hat{s}_2 \\ \hat{s}_3 \end{pmatrix} = \frac{1}{\lVert H \rVert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} - h_{3,j} r^*_{3,j} - h^*_{4,j} r_{4j} \\ h^*_{2,j} r_{1j} - h_{1,j} r^*_{2,j} + h^*_{4,j} r_{3,j} - h_{3,j} r^*_{4,j} \\ h^*_{3,j} r_{1j} + h^*_{4,j} r_{2,j} + h_{1,j} r^*_{3,j} + h_{2,j} r^*_{4,j} \end{pmatrix}.$

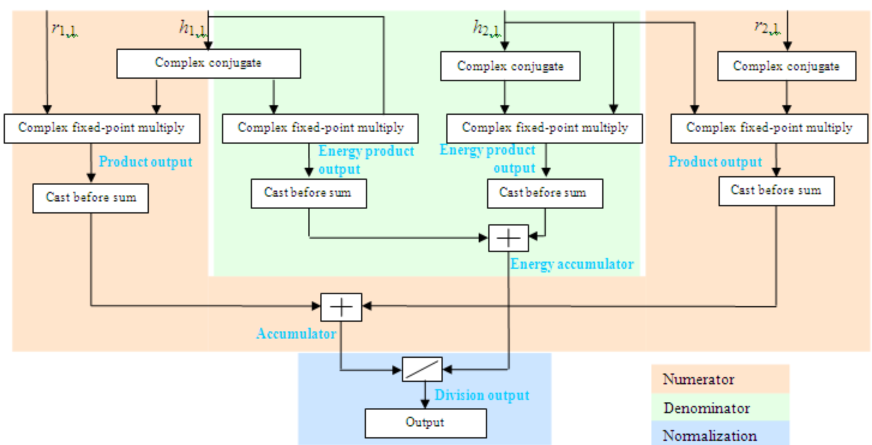
$\hat{s}_k$ represents the estimated kth symbol in the OSTBC codeword matrix. h_{ij} represents the estimate for the channel from the ith transmit antenna and the jth receive antenna. The values of i and j can range from 1 to N (the number of transmit antennas) and to M (the number of receive antennas) respectively. r_{lj} represents the lth symbol at the jth receive antenna per codeword block. The value of l can range from 1 to the codeword block length. $\|H\|^2$ represents the summation of channel power per link, i.e., $\|H\|^2 = \sum_{i=1}^{N}\sum_{j=1}^{M}\|h_{ij}\|^2$

Fixed-Point Signals

Use the following formula for $\hat{s}_1$ for Alamouti code with 1 receive antenna to highlight the data types used for fixed-point signals.

$$\hat{s}_1 = \frac{h^*_{1,1,}\ r_{1,1} + h_{2,1,}r^*_{2,1}}{\|H\|^2} = \frac{h^*_{1,1,}\ r_{1,1} + h_{2,1,}r^*_{2,1}}{h_{1,1}h^*_{1,1,} + h_{2,1,}h^*_{2,1}}$$

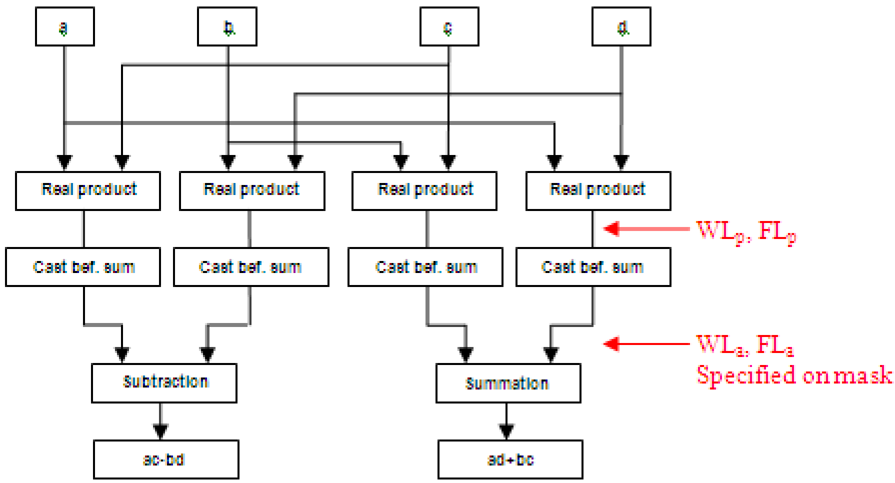
In this equation, the data types for **Product output** and **Accumulator** correspond to the product and summation in the numerator. Similarly, the types for **Energy product output** and **Energy accumulator** correspond to the product and summation in the denominator.



Signal Flow Diagram for s_1 Combining Calculation of Alamouti Code with One Receive Antenna

The following formula shows the data types used within the OSTBC Combiner block for fixed-point signals for more than one receive antenna for Alamouti code, where *M* represents the number of receive antennas.

$$\hat{s}_1 = \frac{h^*_{1,1},\ r_{1,1} + h_{2,1},r^*_{2,1} + h^*_{1,2}\ r_{1,2} + h_{2,2}r^*_{2,2} + \ldots + h^*_{1,M},\ r_{1,M} + h_{2,M}\,r^*_{2,M}}{h_{1.1}h^*_{1.1} + h_{2.1}h^*_{2.1} + h_{1.2}h^*_{1.2} + h_{2.2}h^*_{2.2} + \ldots + h_{1.M}h^*_{1.M} + h_{2.M.}h^*_{2.M}}$$



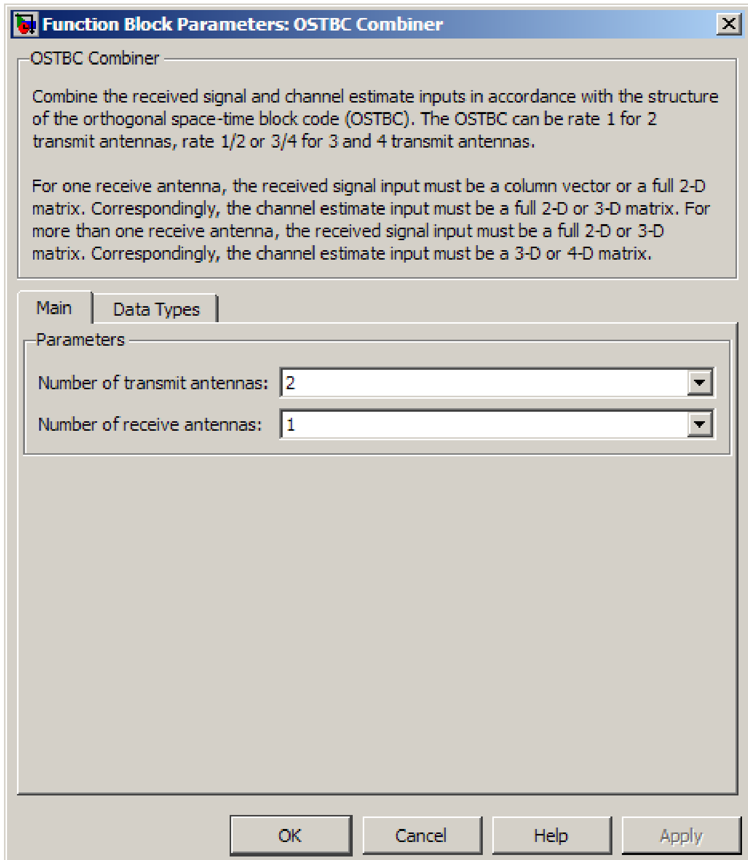
Signal Flow Diagram for Complex Multiply of *a* + *ib* and *c* + *id*

For Binary point scaling, you can not specify WL_p and FL_p. Instead, the blocks determine these values implicitly from WL_a and FL_a

The Internal Rule for **Product output** and **Energy product output** are:

- When you select `Inherit via internal rule`, the internal rule determines WL_p and FL_p. Therefore, $WL_a = WL_p + 1$ and $FL_a = FL_p$
- For `Binary point scaling`, you specify WL_a and FL_a. Therefore, $WL_p = WL_a - 1$ and $FL_a = FL_p$.

For information on how the Internal Rule applies to the **Accumulator** and **Energy Accumulator**, see Inherit via Internal Rule in the *DSP System Toolbox User's Guide*.



Dialog Box

Block Parameters

Number of transmit antennas

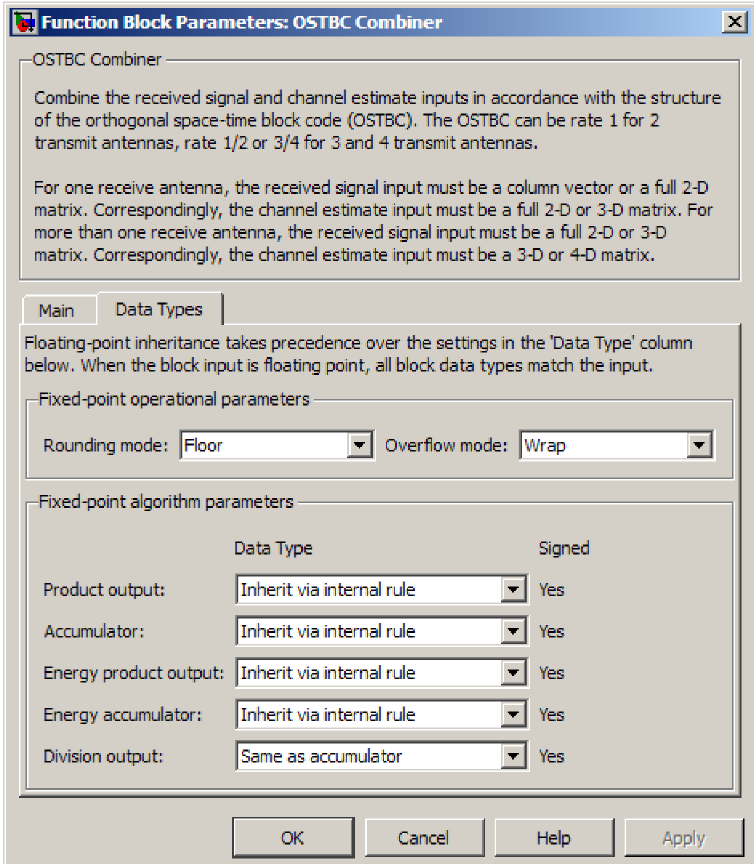
Sets the number of transmit antennas. The block supports 2, 3, or 4 transmit antennas. This value defaults to 2.

Rate

Sets the symbol rate of the code. You can specify either 3/4 or 1/2. This field only appears when you use more than 2 transmit antennas. This field defaults to $\frac{3}{4}$ for more than 2 transmit antennas. For 2 transmit antennas, there is no rate option and the implicit (default) rate defaults to 1.

Number of receive antennas

The number of antennas the block uses to receive signal streams. The block supports from 1 to 8 receive antennas. This value defaults to 1.



Rounding mode

Sets the rounding mode for fixed-point calculations. The block uses the rounding mode if a value cannot be represented exactly by the specified data type and scaling. When this occurs, the value is rounded to a representable number. For more information refer to Rounding in Fixed-Point Designer.

Overflow mode

Sets the overflow mode for fixed-point calculations. Use this parameter to specify the method to be used if the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result. For more information refer to Precision and Range in the Precision and Range section of the *DSP System Toolbox User's Guide*.

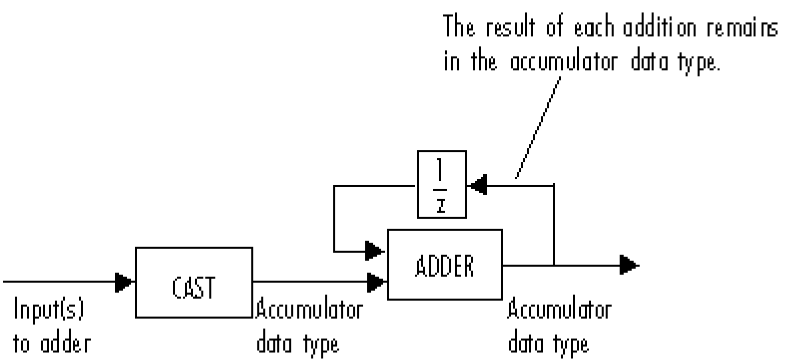
Product Output

Complex product in the numerator for the diversity combining. For more information refer to the Fixed-Point Signals section of this help page.

Accumulator

Summation in the numerator for the diversity combining.

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



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

- When you select `Inherit via internal rule`, the accumulator output word and fraction lengths are automatically calculated for you. Refer to Inherit via Internal Rule for more information.
- 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 first input to the block.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. The bias of all signals in DSP System Toolbox software is zero.

Energy product output
: Complex product in the denominator for calculating total energy in the MIMO channel .

Energy accumulator
: Summation in the denominator for calculating total energy in the MIMO channel.

Division output
: Normalized diversity combining by total energy in the MIMO channel.

Supported Data Type

Port	Supported Data Types
Rx	• Double-precision floating point • Single-precision floating point • Signed Fixed-point
cEst	• Double-precision floating point • Single-precision floating point • Signed Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed Fixed-point

Examples

For an example of this block in use, see OSTBC Over 3x2 Rayleigh Fading Channel in the Communications System Toolbox documentation. The model shows the use of a rate ¾ OSTBC for 3 transmit and 2 receive antennas with BPSK modulation using independent fading links and AWGN.

You can also see the block in the Concatenated OSTBC with TCM example by typing `commtcmostbc` or the IEEE 802.16–2004 OFDM PHY Link, Including Space-Time Block Coding example by typing `commwman80216dstbc` at the MATLAB command line.

See Also

OSTBC Encoder

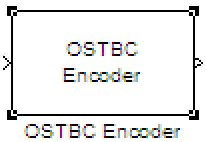
Purpose

Encode input message using orthogonal space-time block code (OSTBC)

Library

MIMO

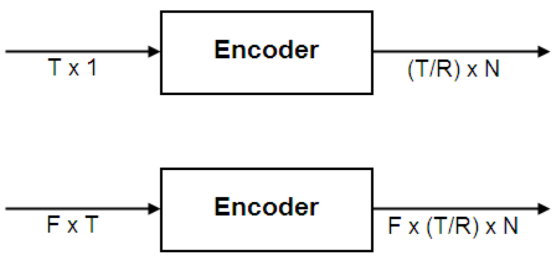
Description



The OSTBC Encoder block encodes an input symbol sequence using orthogonal space-time block code (OSTBC). The block maps the input symbols block-wise and concatenates the output codeword matrices in the time domain. For more information, see the OSTBC Encoding Algorithms section of this help page.

Dimension

The block supports time and spatial domains for OSTBC transmission. It also supports an optional dimension, over which the encoding calculation is independent. This dimension can be thought of as the frequency domain. The following illustration indicates the supported dimensions for the inputs and output of the OSTBC Encoder block.



The following table describes the variables.

Variable	Description
F	The additional dimension; typically the frequency domain. The encoding does not depend on this dimension.
T	Input symbol sequence length for the time domain.

Variable	Description
R	Symbol rate of the code.
N	Number of transmit antennas.

Note On the output, T/R is the symbol sequence length in time domain.

F can be any positive integer. *N* can be 2, 3 or 4, indicated by **Number of transmit antennas**. For $N = 2$, R must be 1. For $N = 3$ or 4, R can be 3/4 or 1/2, indicated by **Rate**. The time domain length T must be a multiple of the number of symbols in each codeword matrix. Specifically, for $N = 2$ or $R = 1/2$, T must be a multiple of 2 and when $R = 3/4$, T must be a multiple of 3.

To understand the block's dimension propagation, refer to the following table.

Dimension	Input	Output
F = 1	Column vector	2-D
F > 1	2-D	3-D

Data Type

For information about the data types each block port supports, see the "Supported Data Type" on page 2-692 table on this page. The output signal inherits the data type from the input signal. For fixed-point signals, the complex conjugation may cause overflows which the fixed-point parameter **Overflow mode** must handle.

Frames

The output signal inherits frame type from the input signal. A column vector input requires either frame-based or sample-based input; otherwise, the input must be sample-based.

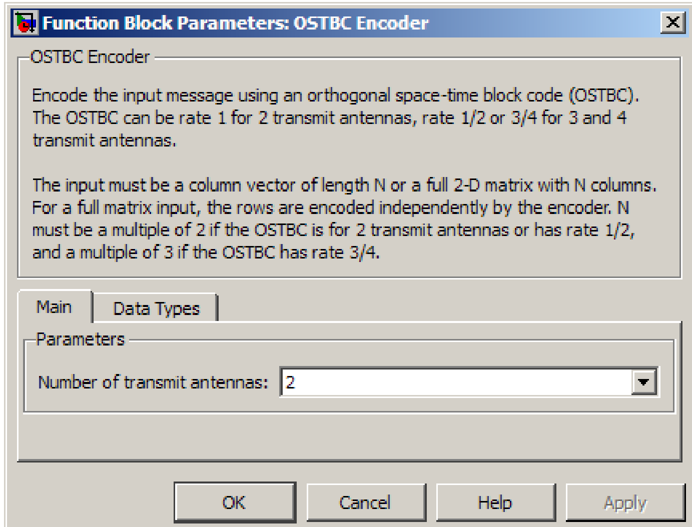
OSTBC Encoding Algorithms

The OSTBC Encoder block supports five different OSTBC encoding algorithms. Depending on the selection for **Rate** and **Number of transmit antennas**, the block implements one of the algorithms in the following table:

Transmit Antenna	Rate	OSTBC Codeword Matrix
2	1	$\begin{pmatrix} s_1 & s_2 \\ -s_2^* & s_1^* \end{pmatrix}$
3	1/2	$\begin{pmatrix} s_1 & s_2 & 0 \\ -s_2^* & s_1^* & 0 \\ 0 & 0 & s_1 \\ 0 & 0 & -s_2^* \end{pmatrix}$
3	3/4	$\begin{pmatrix} s_1 & s_2 & s_3 \\ -s_2^* & s_1^* & 0 \\ s_3^* & 0 & -s_1^* \\ 0 & s_3^* & -s_2^* \end{pmatrix}$

Transmit Antenna	Rate	OSTBC Codeword Matrix
4	1/2	$\begin{pmatrix} s_1 & s_2 & 0 & 0 \\ -s_2^* & s_1^* & 0 & 0 \\ 0 & 0 & s_1 & s_2 \\ 0 & 0 & -s_2^* & s_1^* \end{pmatrix}$
4	3/4	$\begin{pmatrix} s_1 & s_2 & s_3 & 0 \\ -s_2^* & s_1^* & 0 & s_3 \\ s_3^* & 0 & -s_1^* & s_2 \\ 0 & s_3^* & -s_2^* & -s_1 \end{pmatrix}$

In each matrix, its (l, i) entry indicates the symbol transmitted from the ith antenna in the lth time slot of the block. The value of i can range from 1 to N (the number of transmit antennas). The value of l can range from 1 to the codeword block length.



Dialog Box

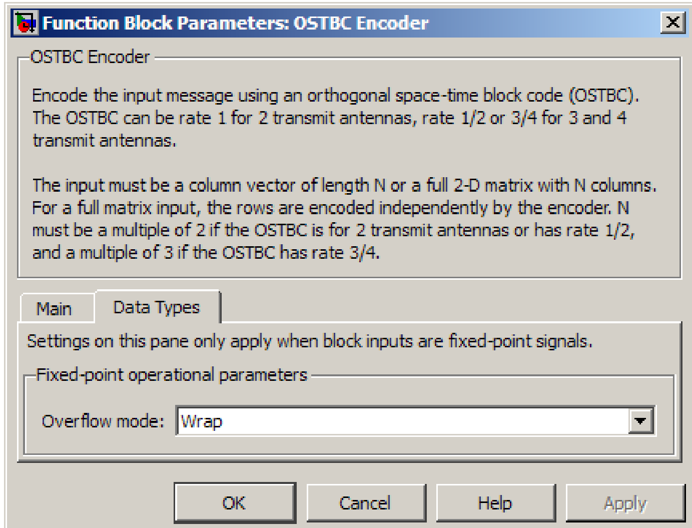
Block Parameters

Number of transmit antennas

Sets the number of antennas at the transmitter side. The block supports 2, 3, or 4 transmit antennas. The value defaults to 2.

Rate

Sets the symbol rate of the code. You can specify either 3/4 or 1/2. This field only appears when using more than 2 transmit antennas. This field defaults to $\frac{3}{4}$ for more than 2 transmit antennas. For 2 transmit antennas, there is no rate option and the rate defaults to 1.



Overflow mode

Sets the overflow mode for fixed-point calculations. Use this parameter to specify the method to be used if the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result. For more information refer to “Precision and Range” in DSP System Toolbox.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Signed Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed Fixed-point

Examples

For an example of this block in use, see OSTBC Over 3x2 Rayleigh Fading Channel in the Communications System Toolbox documentation. The model shows the use of a rate ¾ OSTBC for 3 transmit and 2 receive antennas with BPSK modulation using independent fading links and AWGN

You can also see the block in the Concatenated OSTBC with TCM example by typing `commtcmostbc`. View the IEEE 802.16–2004 OFDM PHY Link, Including Space-Time Block Coding example by typing `commwman80216dstbc` at the MATLAB command line.

See Also

OSTBC Combiner

OVSF Code Generator

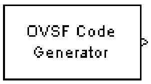
Purpose

Generate orthogonal variable spreading factor (OVSF) code from set of orthogonal codes

Library

Spreading Codes

Description



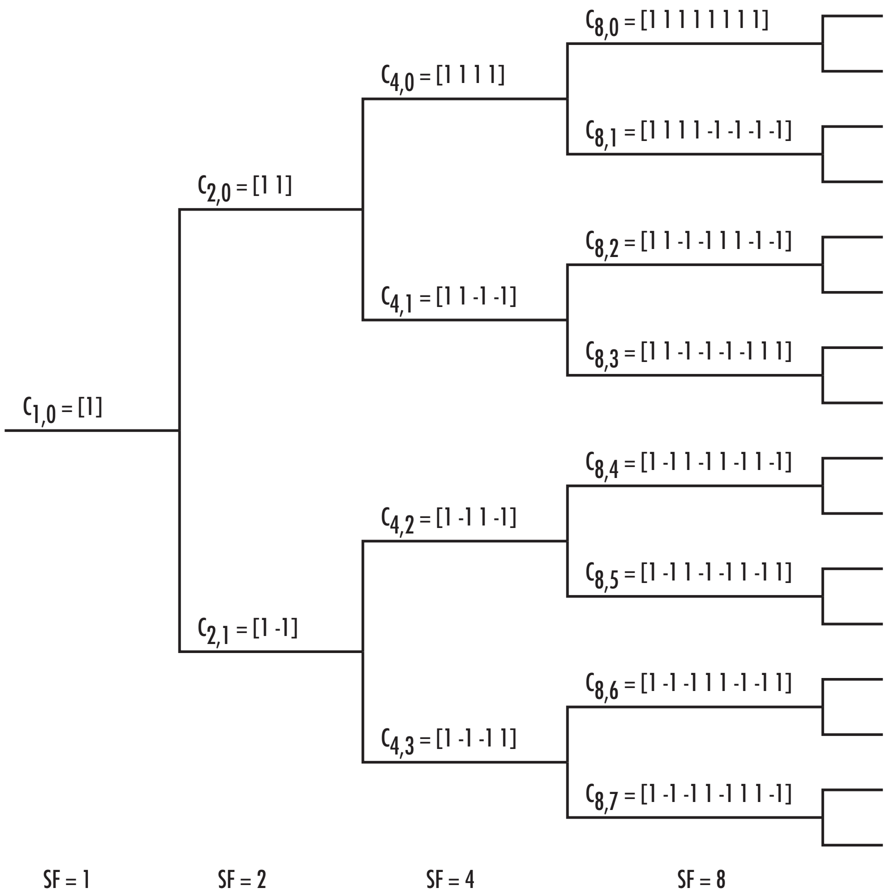
The OVSF Code Generator block generates an OVSF code from a set of orthogonal codes. OVSF codes were first introduced for 3G communication systems. OVSF codes are primarily used to preserve orthogonality between different channels in a communication system.

OVSF codes are defined as the rows of an N-by-N matrix, C_N, which is defined recursively as follows. First, define $C_1 = [1]$. Next, assume that C_N is defined and let $C_N(k)$ denote the kth row of C_N. Define C_{2N} by

$$C_{2N} = \begin{bmatrix} C_N(0) & C_N(0) \\ C_N(0) & -C_N(0) \\ C_N(1) & C_N(1) \\ C_N(1) & -C_N(1) \\ \cdots & \cdots \\ C_N(N-1) & C_N(N-1) \\ C_N(N-1) & -C_N(N-1) \end{bmatrix}$$

Note that C_N is only defined for N a power of 2. It follows by induction that the rows of C_N are orthogonal.

The OVSF codes can also be defined recursively by a tree structure, as shown in the following figure.



If [C] is a code length 2^r at depth r in the tree, where the root has depth 0, the two branches leading out of C are labeled by the sequences [C C] and [C -C], which have length 2^{r+1}. The codes at depth r in the tree are the rows of the matrix C_N, where $N = 2^r$.

Note that two OVSF codes are orthogonal if and only if neither code lies on the path from the other code to the root. Since codes assigned to different users in the same cell must be orthogonal, this restricts the number of available codes for a given cell. For example, if the code C_{41}

in the tree is assigned to a user, the codes C_{10}, C_{20}, C_{82}, C_{83}, and so on, cannot be assigned to any other user in the same cell.

Block Parameters

You specify the code the OVSF Code Generator block outputs by two parameters in the block's dialog: the **Spreading factor**, which is the length of the code, and the **Code index**, which must be an integer in the range [0, 1, ... , N - 1], where N is the spreading factor. If the code appears at depth r in the preceding tree, the **Spreading factor** is 2^r. The **Code index** specifies how far down the column of the tree at depth r the code appears, counting from 0 to N - 1. For $C_{N, k}$ in the preceding diagram, N is the **Spreading factor** and k is the **Code index**.

You can recover the code from the **Spreading factor** and the **Code index** as follows. Convert the **Code index** to the corresponding binary number, and then add 0s to the left, if necessary, so that the resulting binary sequence $x_1\ x_2 \ldots x_r$ has length r, where r is the logarithm base 2 of the **Spreading factor**. This sequence describes the path from the root to the code. The path takes the upper branch from the code at depth i if $x_i = 0$, and the lower branch if $x_i = 1$.

To reconstruct the code, recursively define a sequence of codes C_i for as follows. Let C_0 be the root [1]. Assuming that C_i has been defined, for $i < r$, define C_{i+1} by

$$C_{i+1} = \begin{cases} C_i C_i & \text{if } x_i = 0 \\ C_i(-C_i) & \text{if } x_i = 1 \end{cases}$$

The code C_N has the specified **Spreading factor** and **Code index**.

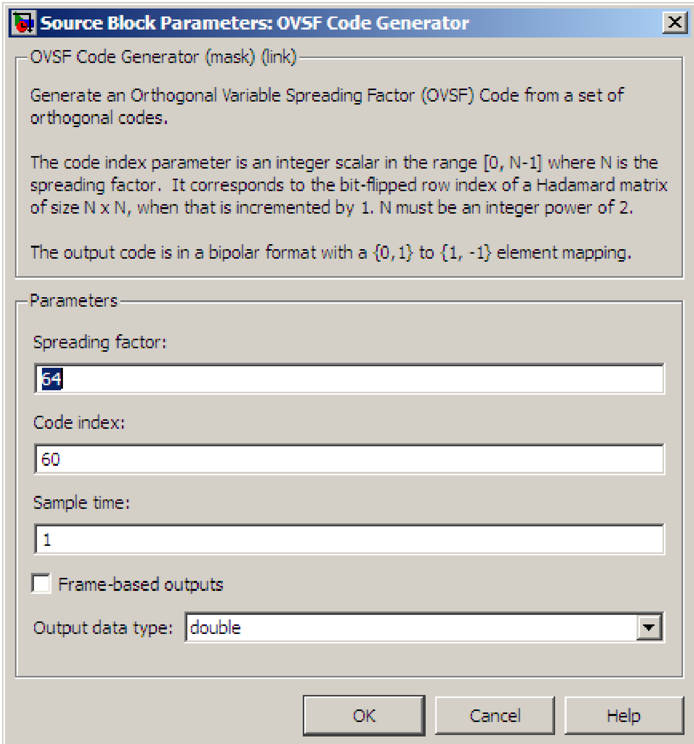
For example, to find the code with **Spreading factor** `16` and **Code index** `6`, do the following:

1. Convert 6 to the binary number 110.
2. Add one 0 to the left to obtain 0110, which has length $4 = \log_2 16$.
3. Construct the sequences C_i according to the following table.

i	x_i	C_i
0		C_0 = [1]
1	0	$C_1 = C_0\ C_0$ = [1] [1]
2	1	$C_2 = C_1\ -C_1$ = [1 1] [-1 -1]
3	1	$C_3 = C_2\ -C_2$ = [1 1 -1 -1] [-1 -1 1 1]
4	0	$C_4 = C_3\ C_3$ = [1 1 -1 -1 -1 -1 1 1] [1 1 -1 -1 -1 -1 1 1]

The code C_4 has **Spreading factor** 16 and **Code index** 6.

Dialog Box



Spreading factor
: Positive integer that is a power of 2, specifying the length of the code.

Code index
: Integer in the range [0, 1, ... , N - 1] specifying the code, where N is the **Spreading factor**.

Sample time
: A positive real scalar specifying the sample time of the output signal.

Frame-based outputs
: Determines whether the output is frame-based or sample-based.

Samples per frame
The number of samples in a frame-based output signal. This field is active only if you select **Frame-based outputs**.

Output data type
The output type of the block can be specified as an `int8` or `double`. By default, the block sets this to `double`.

See Also

Hadamard Code Generator, Walsh Code Generator

Phase/Frequency Offset

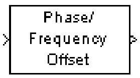
Purpose

Apply phase and frequency offsets to complex baseband signal

Library

RF Impairments

Description



The Phase/Frequency Offset block applies phase and frequency offsets to an incoming signal.

The block inherits its output data type from the input signal. If the input signal is $u(t)$, then the output signal is:

$$y(t) = u(t) \cdot \left(\cos\left(2\pi \int_0^t f(\tau)\, d\tau + \varphi(t) \right) + j \sin\left(2\pi \int_0^t f(\tau)\, d\tau + \varphi(t) \right) \right)$$

where

$f(t)$ = Frequency offset

$\varphi(t)$ = Phase offset

The discrete-time output is:

$$y(0) = u(0)\left(\cos\left(\varphi(0)\right) + j\sin\left(\varphi(0)\right)\right)$$

$$y(i) = u(i)\left(\cos\left(2\pi f\left(i-1\right)\Delta t + \varphi(i)\right) + j\sin\left(2\pi f\left(i-1\right)\Delta t + \varphi(i)\right)\right) \qquad i > 0$$

where

Δt = Sample time

This block accepts real and complex inputs of data type `double` or `single`.

Phase Offset

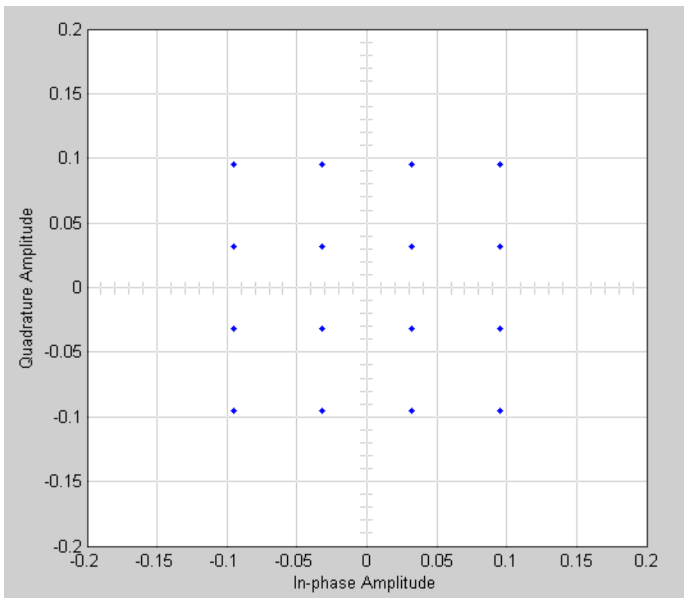
The block applies a phase offset to the input signal, specified by the **Phase offset** parameter.

Frequency Offset

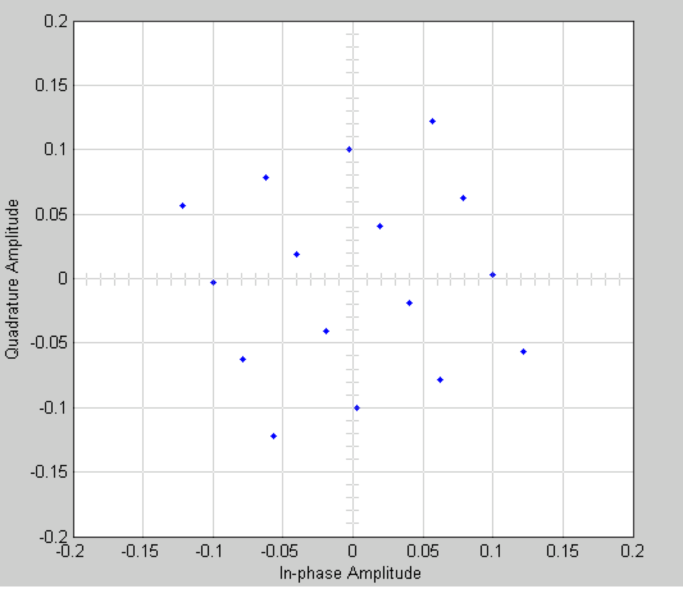
The block applies a frequency offset to the input signal, specified by the **Frequency offset** parameter. Alternatively, when you select

Frequency offset from port, the `Frq` input port provides the offset to the block. The frequency offset must be a scalar value, vector with the same number of rows or columns as the data input, or a matrix with the same size as the data input. For more information, see "Interdependent Parameter-Port Dimensions" on page 2-703.

The effects of changing the block's parameters are illustrated by the following scatter plots of a signal modulated by 16-ary quadrature amplitude modulation (QAM). The usual 16-ary QAM constellation without the effect of the Phase/Frequency Offset block is shown in the first scatter plot:

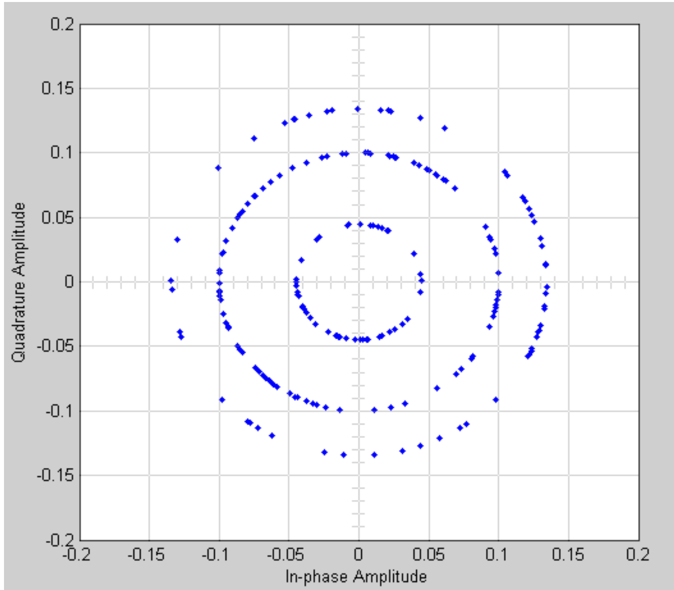


The following figure shows a scatter plot of an output signal, modulated by 16-ary QAM, from the Phase/Frequency Offset block with **Phase offset** set to `20` and **Frequency offset** set to `0`:



Observe that each point in the constellation is rotated by a 20 degree angle counterclockwise.

If you set **Frequency offset** to `2` and **Phase offset** to `0`, the angles of points in the constellation change linearly over time. This causes points in the scatter plot to shift radially, as shown in the following figure:



Note that every point in the scatter plot has magnitude equal to a point in the original constellation.

See "Illustrate RF Impairments That Distort a Signal" for a description of the model that generates this plot.

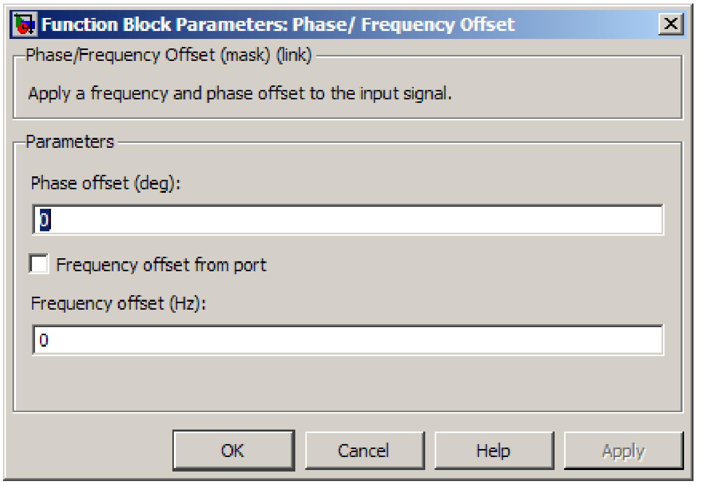
Interdependent Parameter-Port Dimensions

Number of Dimensions	Data I/O Dimension	Frame Size	Number of Channels	Frequency/Phase Offset Parameter Dimension	Frequency Offset Input Port Dimension
Any	Scalar	1	1	Scalar	Scalar
2	M-by-1	M	1	M-by-1, 1-by-M, 1-by-1	M, M-by-1, 1, 1-by-1

Number of Dimensions	Data I/O Dimension	Frame Size	Number of Channels	Frequency/Phase Offset Parameter Dimension	Frequency Offset Input Port Dimension
2	1-by-N	1	N	N-by-1, 1-by-N, 1-by-1	N, 1-by-N, 1, 1-by-1
2	M-by-N	M	N	M-by-N, N-by-1, 1-by-N, M-by-1, 1-by-M, 1-by-1	M-by-N, N, 1-by-N, 1, 1-by-1, M, M-by-1

- When you specify a scalar offset parameter the block applies the same offset to all elements of the input signal
- When you specify a 2-by-1 offset parameter for a 2-by-3 input signal (one offset value per sample), the block applies the same sample offset across the three channels.
- When you specify a 1-by-3 offset parameter for a 2-by-3 input signal (one offset value per channel), the same channel offset is applied across the two samples of a channel.
- When you specify a 2-by-3 offset parameter for a 2-by-3 input signal (one offset value per sample for each channel), the offsets are applied element-wise to the input signal.

Dialog Box



Frequency offset from port

Selecting this option opens a port on the block through which you can input the frequency offset information.

Frequency offset

Specifies the frequency offset in hertz.

This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode. If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. For more information, see Tunable Parameters in the Simulink *User's Guide*.

Phase offset

Specifies the phase offset in degrees.

This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode. If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. For more information, see Tunable Parameters in the Simulink *User's Guide*.

If **Frequency offset** and **Phase offset** are both vectors or both matrices, their dimensions (vector lengths, or number of rows and columns) must be the same.

See Also

Phase Noise

Purpose

Implement phase-locked loop to recover phase of input signal

Library

Components sublibrary of Synchronization

Description

The Phase-Locked Loop (PLL) block is a feedback control system that automatically adjusts the phase of a locally generated signal to match the phase of an input signal. This block is most appropriate when the input is a narrowband signal.

This PLL has these three components:

- A multiplier used as a phase detector.
- A filter. You specify the filter transfer function using the **Lowpass filter numerator** and **Lowpass filter denominator** parameters. Each is a vector that gives the respective polynomial's coefficients in order of descending powers of *s*.

  To design a filter, you can use functions such as `butter`, `cheby1`, and `cheby2` in Signal Processing Toolbox software. The default filter is a Chebyshev type II filter whose transfer function arises from the command below.

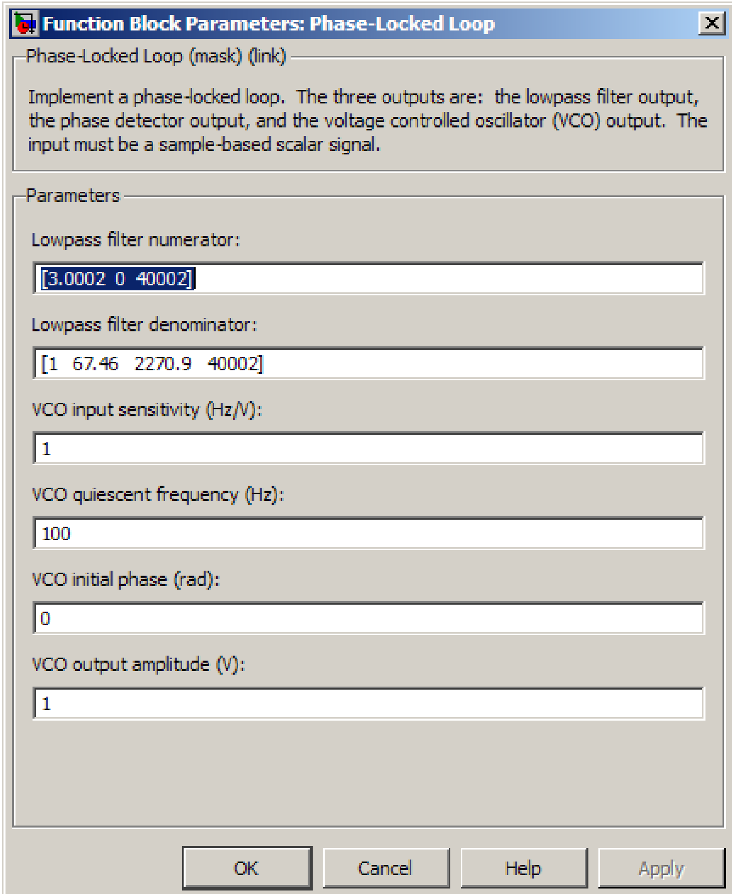
  ```
  [num, den] = cheby2(3,40,100,'s')
  ```

- A voltage-controlled oscillator (VCO). You specify characteristics of the VCO using the **VCO quiescent frequency**, **VCO initial phase**, and **VCO output amplitude** parameters.

This block accepts a sample-based scalar input signal. The input signal represents the received signal. The three output ports produce:

- The output of the filter
- The output of the phase detector
- The output of the VCO

Dialog Box



Lowpass filter numerator

The numerator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of s.

Lowpass filter denominator
The denominator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of s.

VCO input sensitivity (Hz/V)
This value scales the input to the VCO and, consequently, the shift from the **VCO quiescent frequency** value. The units of **VCO input sensitivity** are Hertz per volt.

VCO quiescent frequency (Hz)
The frequency of the VCO signal when the voltage applied to it is zero. This should match the carrier frequency of the input signal.

VCO initial phase (rad)
The initial phase of the VCO signal.

VCO output amplitude
The amplitude of the VCO signal.

See Also

Baseband PLL, Linearized Baseband PLL, Charge Pump PLL

References

For more information about phase-locked loops, see the works listed in "Selected Bibliography for Synchronization" in *Communications System Toolbox User's Guide*.

Phase Noise

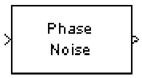
Purpose

Apply receiver phase noise to complex baseband signal

Library

RF Impairments

Description



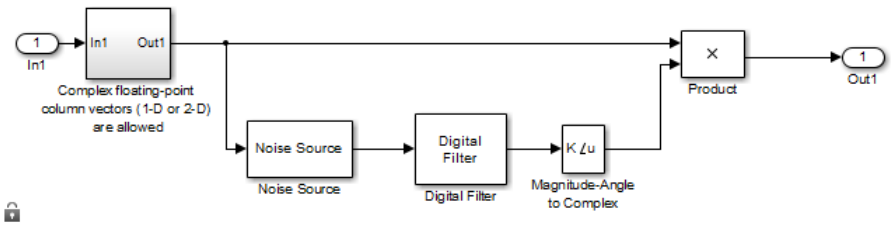
The Phase Noise block applies receiver phase noise to a complex, baseband signal. The block applies the phase noise as follows:

1. Generates additive white Gaussian noise (AWGN) and filters it with a digital filter.
2. Adds the resulting noise to the angle component of the input signal.

Using this block, you can specify phase noise as a scalar frequency offset or a vector of frequency offsets.

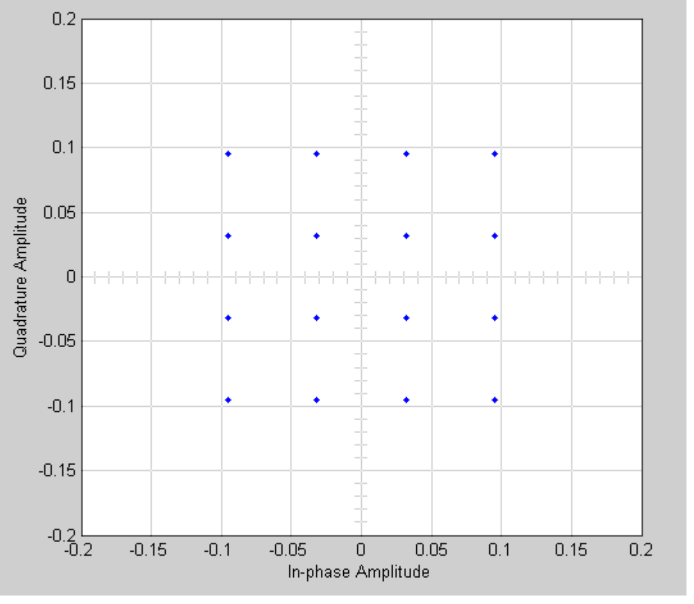
- For a scalar frequency offset, the block generates phase noise over the entire spectral observation window, from 0 Hz (or as close as possible to 0 Hz) to $\pm\frac{F_S}{2}$, where F_s represents the sampling frequency. The noise is scaled so that it is at the block-specific phase noise level at the specified frequency offset. The block generates a phase noise with $\frac{1}{f}$ characteristic over the entire frequency range.
- For a vector of frequency offsets, the block interpolates the spectrum mask across log10(frequency), and is flat from the highest frequency offset to half the sample rate.

You can view the block's implementation of phase noise by right-clicking on the block and selecting **Mask** > **Look under mask**. This displays the following figure:

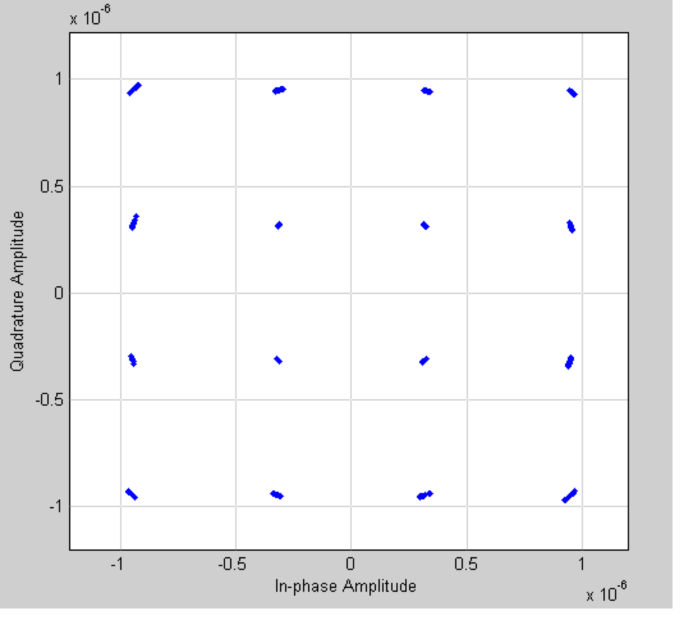


You can view the construction of the Noise Source subsystem by double-clicking it.

The effects of changing the block's parameters are illustrated by the following scatter plots of a signal modulated by 16-ary quadrature amplitude modulation (QAM). The usual 16-ary QAM constellation without distortion is shown in the first scatter plot:

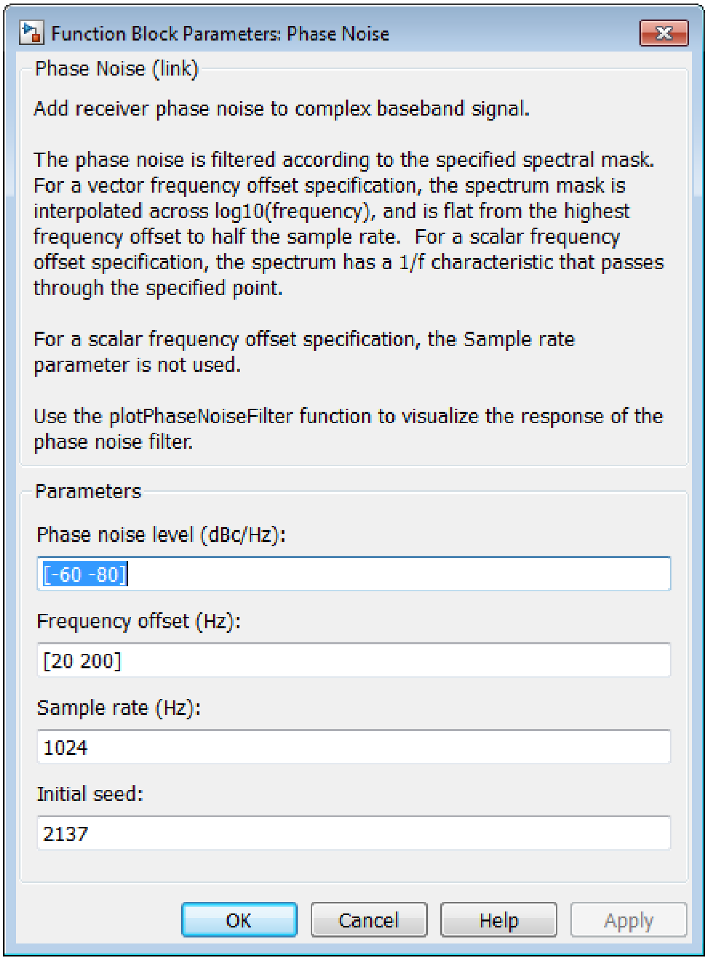


The following figure shows a scatter plot of an output signal, modulated by 16-ary QAM, from the Phase Noise block with **Phase noise level (dBc/Hz)** set to `-70` and **Frequency offset (Hz)** set to `100`:



This plot is generated by the model described in "Illustrate RF Impairments That Distort a Signal" with the following parameter settings for the Rectangular QAM Modulator Baseband block:

- **Normalization method** set to `Average Power`
- **Average power (watts)** set to `1e-12`



Dialog Box

Phase noise level (dBc/Hz)

Scalar or vector that specifies the phase noise level. Specify the phase noise level in decibels relative to carrier per Hertz (dBc/Hz). The lengths of the phase noise level and frequency offset vectors must be equal.

Frequency offset (Hz)
Specifies the frequency offset in Hertz. If the frequency offset is a vector, then the vector must be monotonically increasing. The lengths of the phase noise level and frequency offset vectors must be equal.

Sample rate (Hz)
Must be greater than twice the largest value of the **Frequency offset** vector to avoid aliasing. Specify in Hertz. When you specify a vector of frequency offsets, the block uses this parameter. The block does not use this parameter when you specify a scalar frequency offset.

The sample rate must match the sample rate of the input signal. This quantity is the actual sample rate, not the frame rate of a frame-based signal.

Initial seed
Nonnegative integer specifying the initial seed for the random number generator the block uses to generate noise.

Examples

For an example model that uses this block, see "View Phase Noise Effects on Signal Spectral".

See Also

Phase/Frequency Offset

References

[1] Kasdin, N.J., "Discrete Simulation of Colored Noise and Stochastic Processes and 1/(f^alpha); Power Law Noise Generation," The Proceedings of the IEEE, May, 1995, Vol. 83, No. 5

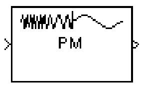
Purpose

Demodulate PM-modulated data

Library

Analog Passband Modulation, in Modulation

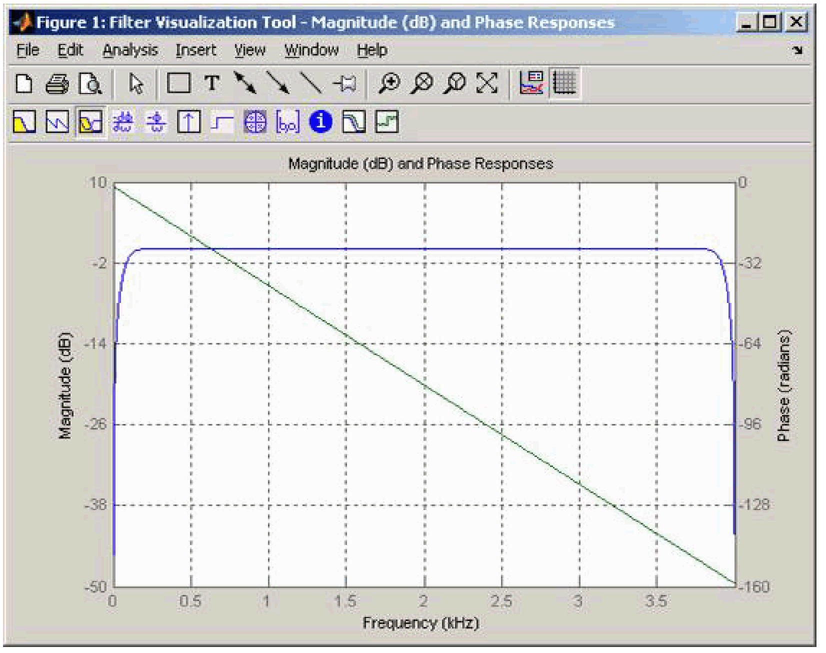
Description



The PM Demodulator Passband block demodulates a signal that was modulated using phase modulation. The input is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

For best results, use a carrier frequency which is estimated to be larger than 10% of your input signal's sample rate. This is due to the implementation of the Hilbert transform by means of a filter.

In the following example, we sample a 10Hz input signal at 8000 samples per second. We then designate a Hilbert Transform filter of order 100. Below is the response of the Hilbert Transform filter as returned by `fvtool`.

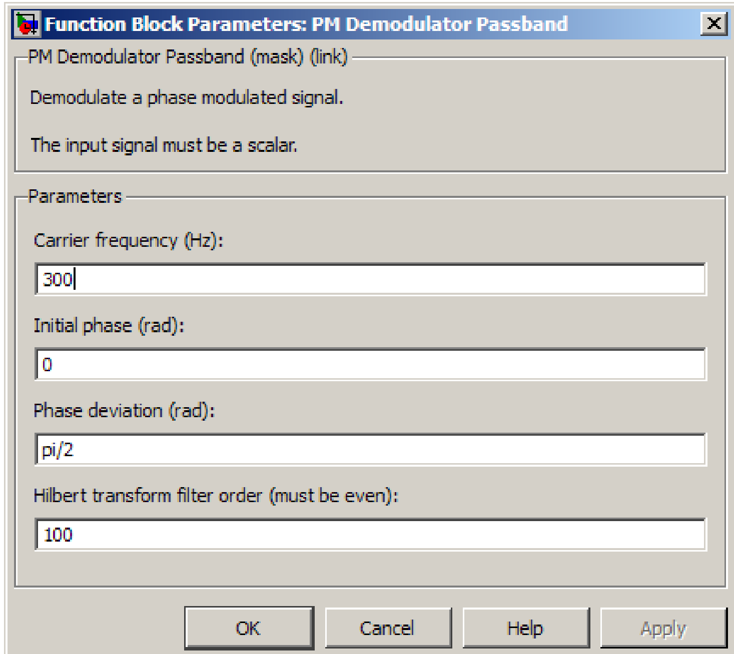


Note the bandwidth of the filter's magnitude response. By choosing a carrier frequency larger than 10% (but less than 90%) of the input signal's sample rate (8000 samples per second, in this example) or equivalently, a carrier frequency larger than 400Hz, we ensure that the Hilbert Transform Filter will be operating in the flat section of the filter's magnitude response (shown in blue), and that our modulated signal will have the desired magnitude and form.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Dialog Box



Carrier frequency (Hz)
: The frequency of the carrier.

Initial phase (rad)
: The initial phase of the carrier in radians.

Phase deviation (rad)
: The phase deviation of the carrier frequency in radians. Sometimes it is referred to as the "variation" in the phase.

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

Pair Block

PM Modulator Passband

PM Modulator Passband

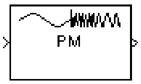
Purpose

Modulate using phase modulation

Library

Analog Passband Modulation, in Modulation

Description



The PM Modulator Passband block modulates using phase modulation. The output is a passband representation of the modulated signal. The output signal's frequency varies with the input signal's amplitude. Both the input and output signals are real scalar signals.

If the input is $u(t)$ as a function of time t, then the output is

$$\cos(2\pi f_c t + K_c u(t) + \theta)$$

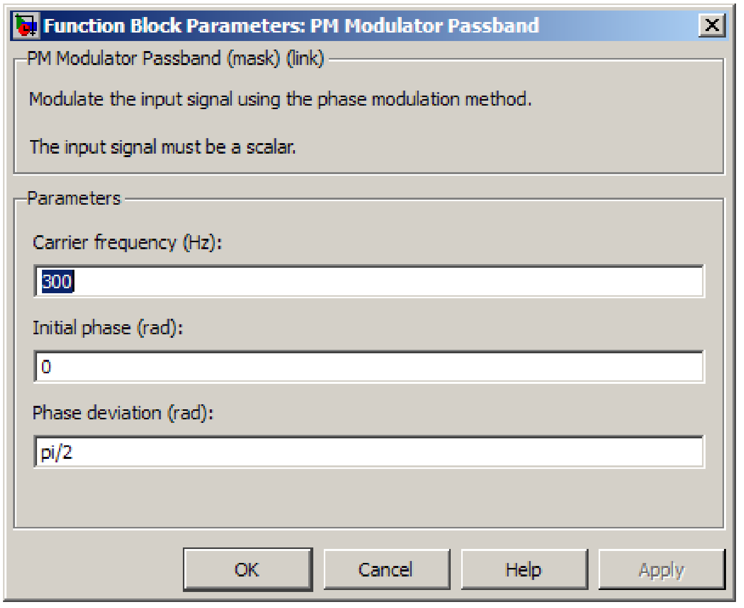
where

- f_c represents the **Carrier frequency** parameter
- θ represents the **Initial phase** parameter
- K_c represents the **Phase deviation** parameter

An appropriate **Carrier frequency** value is generally much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Dialog Box



Carrier frequency (Hz)
The frequency of the carrier.

Initial phase (rad)
The initial phase of the carrier in radians.

Phase deviation (rad)
The phase deviation of the carrier frequency in radians. This is sometimes referred to as the "variation" in the phase.

Pair Block

PM Demodulator Passband

PN Sequence Generator

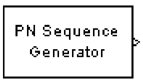
Purpose

Generate pseudonoise sequence

Library

Sequence Generators sublibrary of Comm Sources

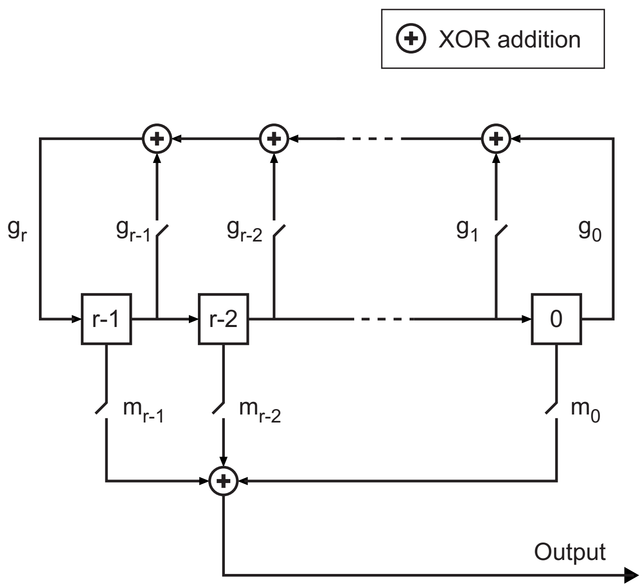
Description



The PN Sequence Generator block generates a sequence of pseudorandom binary numbers using a linear-feedback shift register (LFSR). This block implements LFSR using a simple shift register generator (SSRG, or Fibonacci) configuration. A pseudonoise sequence can be used in a pseudorandom scrambler and descrambler. It can also be used in a direct-sequence spread-spectrum system.

This block can output sequences that vary in length during simulation. For more information about variable-size signals, see “Variable-Size Signal Basics” in the Simulink documentation.

The PN Sequence Generator block uses a shift register to generate sequences, as shown below.



All *r* registers in the generator update their values at each time step, according to the value of the incoming arrow to the shift register. The adders perform addition modulo 2. The shift register is described by the **Generator Polynomial** parameter, which is a primitive binary polynomial in *z*, $g_r z^r + g_{r-1} z^{r-1} + g_{r-2} z^{r-2} + \ldots + g_0$. The coefficient g_k is 1 if there is a connection from the kth register, as labeled in the preceding diagram, to the adder. The leading term g_r and the constant term g_0 of the **Generator Polynomial** parameter must be 1 because the polynomial must be primitive.

You can specify the **Generator polynomial** parameter using either of these formats:

- A vector that lists the coefficients of the polynomial in descending order of powers. The first and last entries must be 1. Note that the length of this vector is one more than the degree of the generator polynomial.
- A vector containing the exponents of *z* for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`.

For example, `[1 0 0 0 0 0 1 0 1]` and `[8 2 0]` represent the same polynomial, $p(z) = z^8 + z^2 + 1$.

The **Initial states** parameter is a vector specifying the initial values of the registers. The **Initial states** parameter must satisfy these criteria:

- All elements of the **Initial states** vector must be binary numbers.
- The length of the **Initial states** vector must equal the degree of the generator polynomial.

Note At least one element of the **Initial states** vector must be nonzero in order for the block to generate a nonzero sequence. That is, the initial state of at least one of the registers must be nonzero.

For example, the following table indicates two sets of parameter values that correspond to a generator polynomial of $p(z) = z^8 + z^2 + 1$.

Quantity	Example 1	Example 2
Generator polynomial	`g1 = [1 0 0 0 0 0 1 0 1]`	`g2 = [8 2 0]`
Degree of generator polynomial	8, which is `length(g1)-1`	8
Initial states	`[1 0 0 0 0 0 1 0]`	`[1 0 0 0 0 0 1 0]`

Output mask vector (or scalar shift value) shifts the starting point of the output sequence. With the default setting for this parameter, the only connection is along the arrow labeled m_0, which corresponds to a shift of 0. The parameter is described in greater detail below.

You can shift the starting point of the PN sequence with **Output mask vector (or scalar shift value)**. You can specify the parameter in either of two ways:

- An integer representing the length of the shift
- A binary vector, called the *mask vector*, whose length is equal to the degree of the generator polynomial

The difference between the block's output when you set **Output mask vector (or scalar shift value)** to 0, versus a positive integer d, is shown in the following table.

	T = 0	T = 1	T = 2	...	T = d	T = d+1
Shift = 0	x_0	x_1	x_2	...	x_d	x_{d+1}
Shift = d	x_d	x_{d+1}	x_{d+2}	...	x_{2d}	x_{2d+1}

Alternatively, you can set **Output mask vector (or scalar shift value)** to a binary vector, corresponding to a polynomial in *z,* $m_{r-1}z^{r-1} + m_{r-2}z^{r-2} + ... + m_1z + m_0$, of degree at most r-1. The mask vector corresponding to a shift of d is the vector that represents $m(z) = z^d$ modulo g(*z*), where g(*z*) is the generator polynomial. For example, if the degree of the generator polynomial is 4, then the mask vector corresponding to d = 2 is `[0 1 0 0]`, which represents the polynomial $m(z) = z^2$. The preceding schematic diagram shows how **Output mask vector (or scalar shift value)** is implemented when you specify it as a mask vector. The default setting for **Output mask vector (or scalar shift value)** is `0`. You can calculate the mask vector using the Communications System Toolbox function `shift2mask`.

You can use an external signal to reset the values of the internal shift register to the initial state by selecting **Reset on nonzero input**. This creates an input port for the external signal in the PN Sequence Generator block. The way the block resets the internal shift register depends on whether its output signal and the reset signal are sample-based or frame-based. The following example demonstrates the possible alternatives.

Example: Resetting a Signal

Suppose that the PN Sequence Generator block outputs `[1 0 0 1 1 0 1 1]` when there is no reset. You then select **Reset on nonzero input** and input a reset signal [0 0 0 1]. The following table shows three possibilities for the properties of the reset signal and the PN Sequence Generator block.

Reset Signal Properties	PN Sequence Generator block	Reset Signal, Output Signal
Sample-based **Sample time** = 1	Sample-based **Sample time** = 1	Reset 0 0 0 $\boxed{1}$ 1 0 0 $\boxed{1}$ 0 0 1 1 0 1 1
Frame-based **Sample time** =1 **Samples per frame** = 2	Frame-based **Sample time** = 1 **Samples per frame** = 2	Reset 0 0 0 $\boxed{1}$ 1 0 0 $\boxed{1}$ 0 0 1 1 0 1 1
Sample-based **Sample time** = 2 **Samples per frame** = 1	Frame-based **Sample time** = 1 **Samples per frame** = 2	Reset 0 0 0 $\boxed{1}$ 1 0 0 1 0 0 $\boxed{1}$ 0 0 1 1 0 1 1

In the first two cases, the PN sequence is reset at the fourth bit, because the fourth bit of the reset signal is a 1 and the **Sample time** is 1. Note that in the second case, the frame sizes are 2, and the reset occurs at the end of the second frame.

In the third case, the PN sequence is reset at the seventh bit. This is because the reset signal has **Sample time** 2, so the reset bit is first sampled at the seventh bit. With these settings, the reset always occurs at the beginning of a frame.

Attributes of Output Signal

If the **Frame-based outputs** box is selected, the output signal is a frame-based column vector whose length is the **Samples per frame** parameter. Otherwise, the output signal is a one-dimensional scalar.

Sequences of Maximum Length

If you want to generate a sequence of the maximum possible length for a fixed degree, *r*, of the generator polynomial, you can set

Generator polynomial to a value from the following table. See [1] for more information about the shift-register configurations that these polynomials represent.

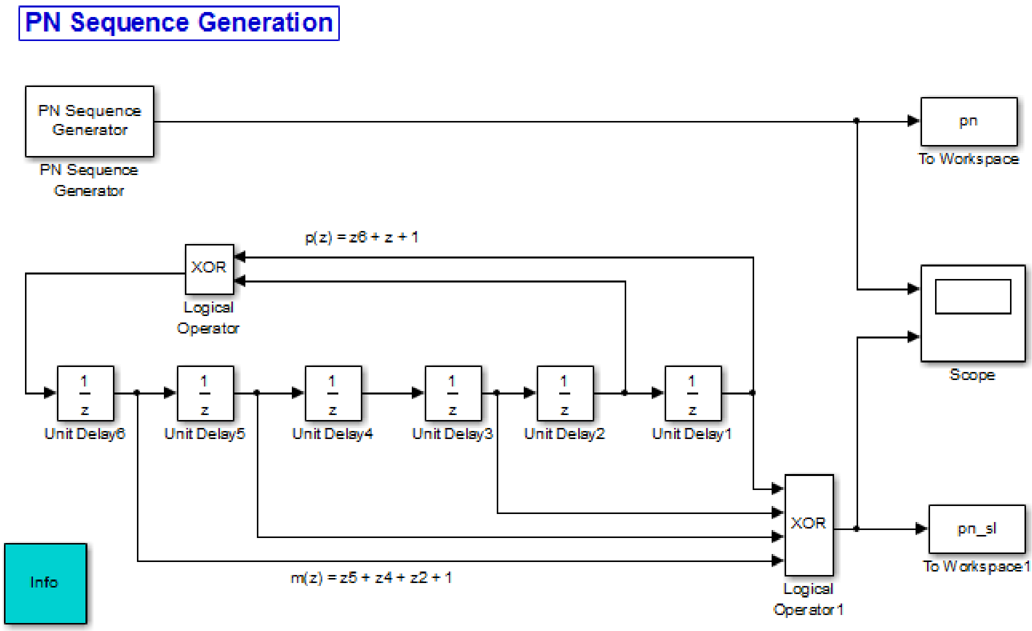
r	Generator Polynomial	r	Generator Polynomial
2	[2 1 0]	21	[21 19 0]
3	[3 2 0]	22	[22 21 0]
4	[4 3 0]	23	[23 18 0]
5	[5 3 0]	24	[24 23 22 17 0]
6	[6 5 0]	25	[25 22 0]
7	[7 6 0]	26	[26 25 24 20 0]
8	[8 6 5 4 0]	27	[27 26 25 22 0]
9	[9 5 0]	28	[28 25 0]
10	[10 7 0]	29	[29 27 0]
11	[11 9 0]	30	[30 29 28 7 0]
12	[12 11 8 6 0]	31	[31 28 0]
13	[13 12 10 9 0]	32	[32 31 30 10 0]
14	[14 13 8 4 0]	33	[33 20 0]
15	[15 14 0]	34	[34 15 14 1 0]
16	[16 15 13 4 0]	35	[35 2 0]
17	[17 14 0]	36	[36 11 0]
18	[18 11 0]	37	[37 12 10 2 0]
19	[19 18 17 14 0]	38	[38 6 5 1 0]
20	[20 17 0]	39	[39 8 0]
40	[40 5 4 3 0]	47	[47 14 0]
41	[41 3 0]	48	[48 28 27 1 0]

r	Generator Polynomial	r	Generator Polynomial
42	[42 23 22 1 0]	49	[49 9 0]
43	[43 6 4 3 0]	50	[50 4 3 2 0]
44	[44 6 5 2 0]	51	[51 6 3 1 0]
45	[45 4 3 1 0]	52	[52 3 0]
46	[46 21 10 1 0]	53	[53 6 2 1 0]

Example of PN Sequence Generation

This example clarifies the operation of the `PN Sequence Generator` block by comparing the output sequence from the library block with that generated from primitive Simulink blocks.

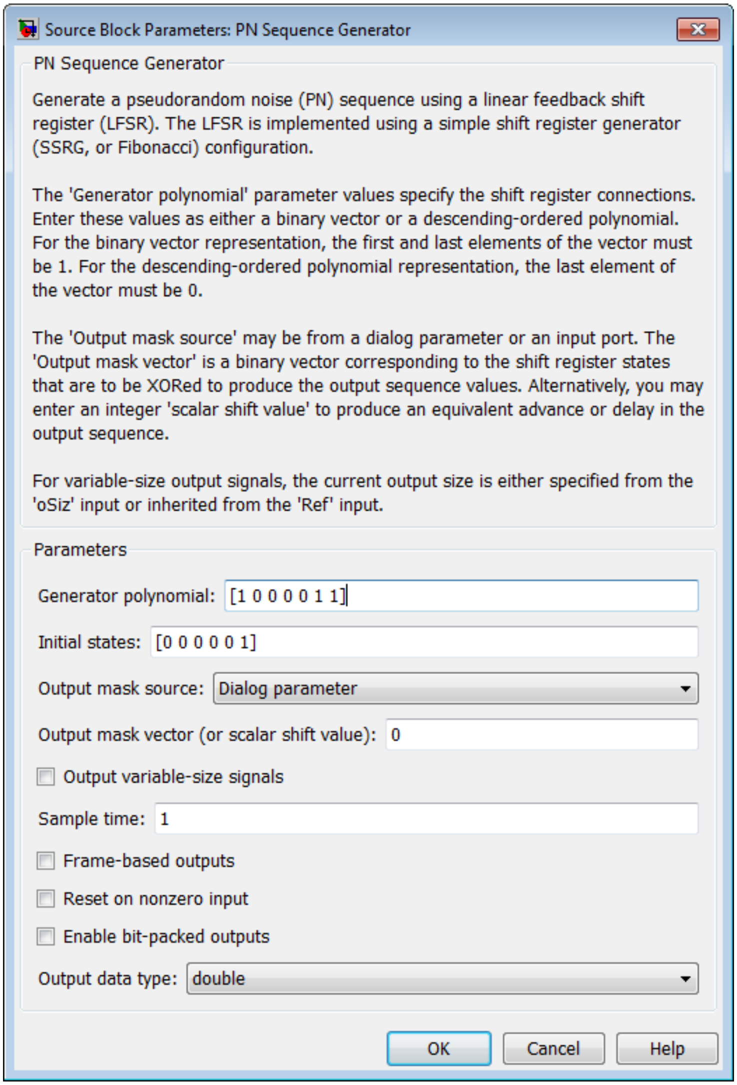
To open the model, enter `doc_pnseq2` at the MATLAB command line.



For the chosen generator polynomial, $p(z) = z^6 + z + 1$, the model generates a PN sequence of period 63, using both the library block and corresponding Simulink blocks. It shows how the two parameters, **Initial states** and **Output mask vector (or scalar shift value)**, are interpreted in the latter schematic.

You can experiment with different initial states, by changing the value of **Initial states** prior to running the simulation. For all values, the two generated sequences are the same.

Using the PN Sequence Generator block allows you to easily generate PN sequences of large periods.



Dialog Box

Generator polynomial

Polynomial that determines the shift register's feedback connections.

Initial states
Vector of initial states of the shift registers.

Output mask source
Specifies how output mask information is given to the block.

- When you set this parameter to `Dialog parameter`, the field **Output mask vector (or scalar shift value)** is enabled for user input.
- When set this parameter to `Input port`, a `Mask` input port appears on the block icon. The `Mask` input port only accepts mask vectors.

Output mask vector (or scalar shift value)
This field is available only when **Output mask source** is set to `Dialog parameter`.

Integer scalar or binary vector that determines the delay of the PN sequence from the initial time. If you specify the shift as a binary vector, the vector's length must equal the degree of the generator polynomial.

Output variable-size signals
Select this check box if you want the output sequences to vary in length during simulation. The default selection outputs fixed-length signals.

Maximum output size source
Specify how the block defines maximum output size for a signal.

- When you select `Dialog parameter`, the value you enter in the **Maximum output size** parameter specifies the maximum size of the output. When you make this selection, the `oSiz` input port specifies the current size of the output signal and the block output inherits sample time from the input signal. The input value must be less than or equal to the **Maximum output size** parameter.

- When you select `Inherit from reference port`, the block output inherits sample time, maximum size, and current size from the variable-sized signal at the Ref input port.

This parameter only appears when you select **Output variable-size signals**. The default selection is `Dialog parameter`.

Maximum output size
: Specify a two-element row vector denoting the maximum output size for the block. The second element of the vector must be `1` For example, [10 1] gives a 10-by-1 maximum sized output signal. This parameter only appears when you select **Output variable-size signals**.

Sample time
: Period of each element of the output signal.

Frame-based outputs
: Determines whether the output is frame-based or sample-based.

Samples per frame
: The number of samples in a frame-based output signal. This field is active only if you select **Frame-based outputs**.

Reset on nonzero input
: When selected, you can specify an input signal that resets the internal shift registers to the original values of the **Initial states** parameter.

Enable bit-packed outputs
: When selected, the field **Number of packed bits** and the option **Interpret bit-packed values as signed** is enabled.

Number of packed bits
: Indicates how many bits to pack into each output data word (allowable range is 1 to 32).

Interpret bit-packed values as signed

Indicates whether packed bits are treated as signed or unsigned integer data values. When selected, a 1 in the most significant bit (sign bit) indicates a negative value.

Output data type

By default, this is set to `double`.

When **Enable bit-packed outputs** is not selected, the output data type can be specified as a `double`, `boolean`, or `Smallest unsigned integer`. When the parameter is set to `Smallest unsigned integer`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type is the ideal minimum one-bit size, i.e., `ufix(1)`. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding to the size of a char (e.g., `uint8`).

When **Enable bit-packed outputs** is selected, the output data type can be specified as `double` or `Smallest integer`. When the parameter is set to `Smallest integer`, the output data type is selected based on **Interpret bit-packed values as signed**, **Number of packed bits**, and the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type is the ideal minimum `n`-bit size, i.e., `sfix(n)` or `ufix(n)`, based on **Interpret bit-packed values as signed**. For all other selections, it is a signed or unsigned integer with the smallest available word length large enough to fit `n` bits.

See Also

Kasami Sequence Generator, Scrambler

References

[1] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

[2] Lee, J. S., and L. E. Miller, *CDMA Systems Engineering Handbook*, Artech House, 1998.

[3] Golomb, S.W., *Shift Register Sequences*, Aegean Park Press, 1967.

Poisson Integer Generator

Purpose

Generate Poisson-distributed random integers

Library

Random Data Sources sublibrary of Comm Sources

Description

The Poisson Integer Generator block generates random integers using a Poisson distribution. The probability of generating a nonnegative integer k is

$$\lambda^k \exp(-\lambda)/(k!)$$

where λ is a positive number known as the Poisson parameter.

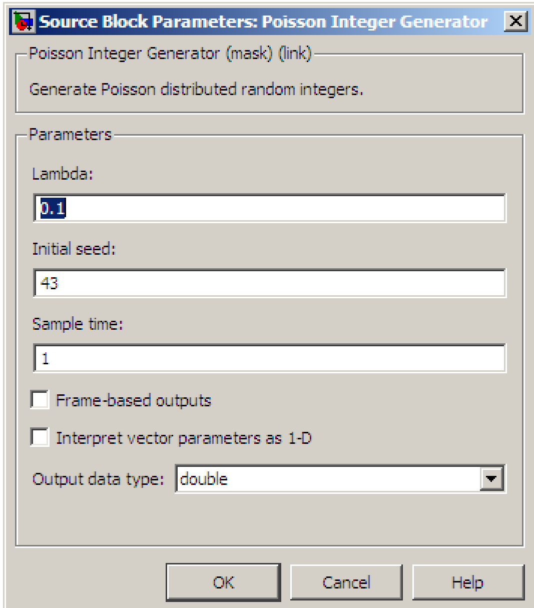
You can use the Poisson Integer Generator to generate noise in a binary transmission channel. In this case, the Poisson parameter **Lambda** should be less than 1, usually much less.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples per frame**, and **Interpret vector parameters as 1-D** parameters. See "Sources and Sinks" in *Communications System Toolbox User's Guide* for more details.

The number of elements in the **Initial seed** parameter becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. Also, the shape (row or column) of the **Initial seed** parameter becomes the shape of a sample-based two-dimensional output signal.

Dialog Box



Lambda

The Poisson parameter λ. If it is a scalar, then every element in the output vector shares the same Poisson parameter.

Initial seed

The initial seed value for the random number generator.

Sample time

The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs

Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame

The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D

If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type

The output type of the block can be specified as a `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`. By default, the block sets this to `double`.

See Also

Random Integer Generator; `poissrnd` (Statistics Toolbox)

Puncture

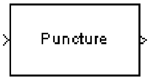
Purpose

Output elements which correspond to 1s in binary Puncture vector

Library

Sequence Operations

Description



The Puncture block creates an output vector by removing selected elements of the input vector and preserving others. This block accepts an input signal that is a real or complex vector of length K. The block determines which elements to remove and preserve by using the binary **Puncture vector** parameter.

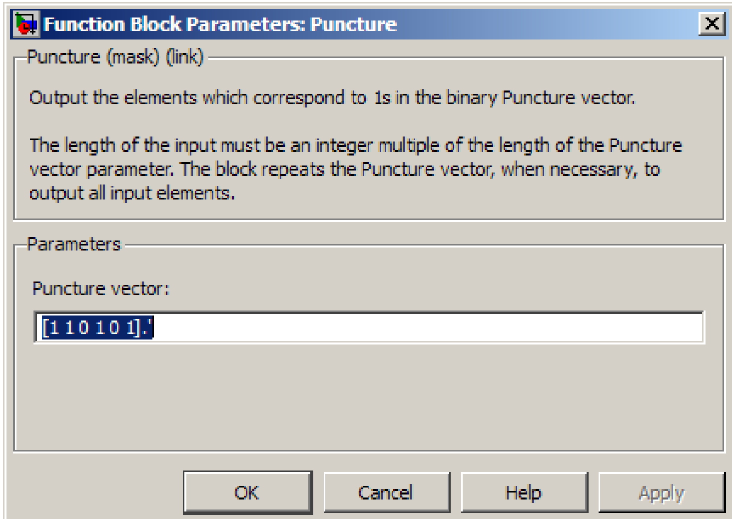
and mod is the modulus function (`mod` in MATLAB).

- If **Puncture vector**(n) = 0, then the block removes the n^{th} element of the input vector and does not include it as part of the output vector.
- If **Puncture vector**(n) = 1, then the block preserves the n^{th} element of the input vector as part of the output vector.

The input length, K, must be an integer multiple of the **Puncture vector** parameter length. The block repeats the puncturing pattern, as necessary, to include all input elements. The preserved elements appear in the output vector in the same order in which they appear in the input vector.

The input signal and the puncture vector are both column vectors.

The block accepts signals with the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.



Dialog Box

Puncture vector

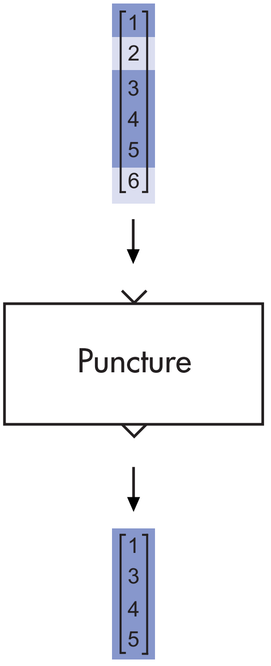
A binary vector whose pattern of 0s (1s) indicates which elements of the input the block should remove (preserve).

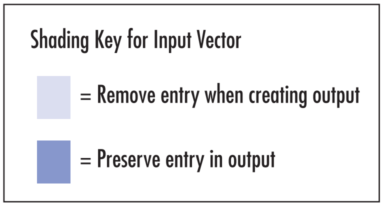
Examples

If the **Puncture vector** parameter is the six-element vector `[1;0;1;1;1;0]`, then the block:

- Removes the second and sixth elements from the group of six input elements.
- Sends the first, third, fourth, and fifth elements to the output vector.

The diagram below depicts the block's operation on an input vector of `[1;2;3;4;5;6]`, using this **Puncture vector** parameter.





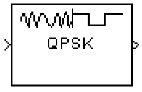
See Also

Insert Zero

Purpose Demodulate QPSK-modulated data

Library PM, in Digital Baseband sublibrary of Modulation

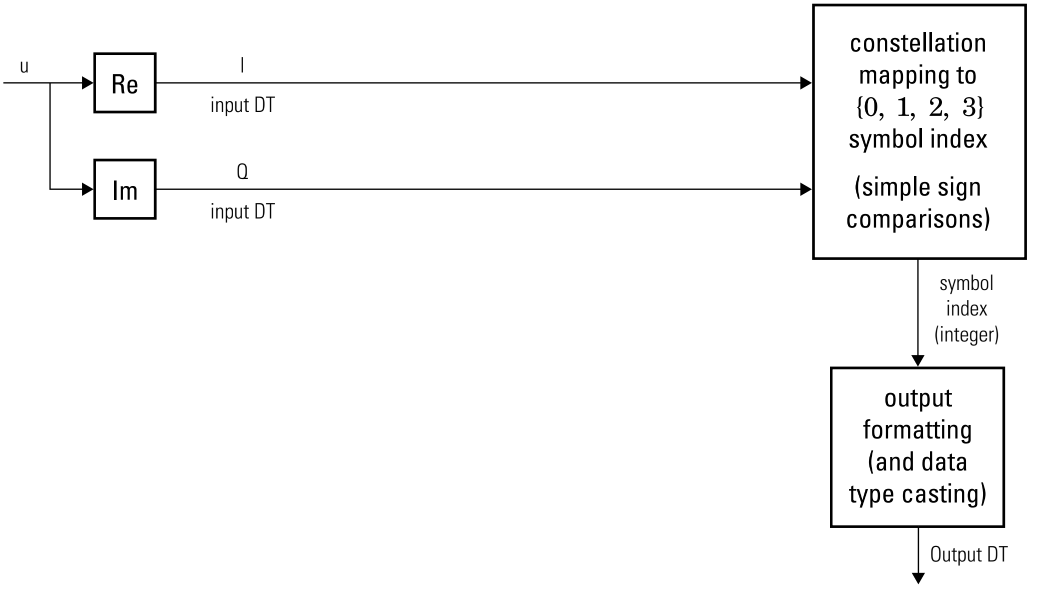
Description



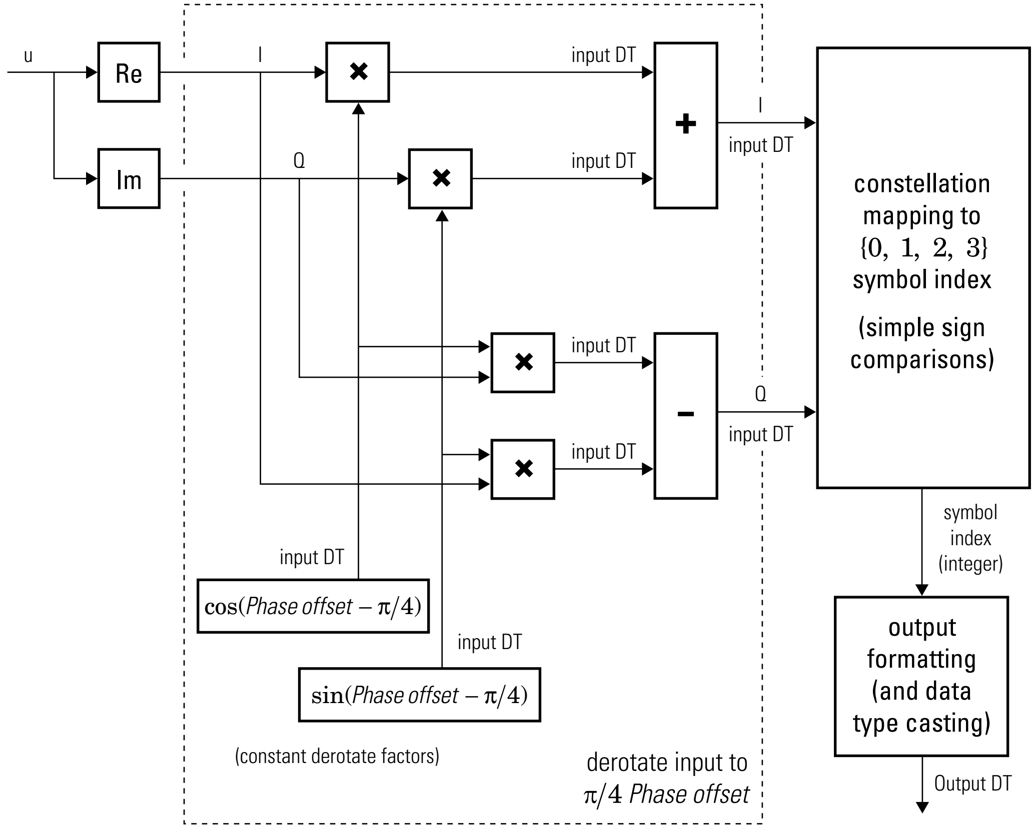
The QPSK Demodulator Baseband block demodulates a signal that was modulated using the quaternary phase shift keying method. The input is a baseband representation of the modulated signal.

The input must be a complex signal. This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see "Supported Data Types" on page 2-747.

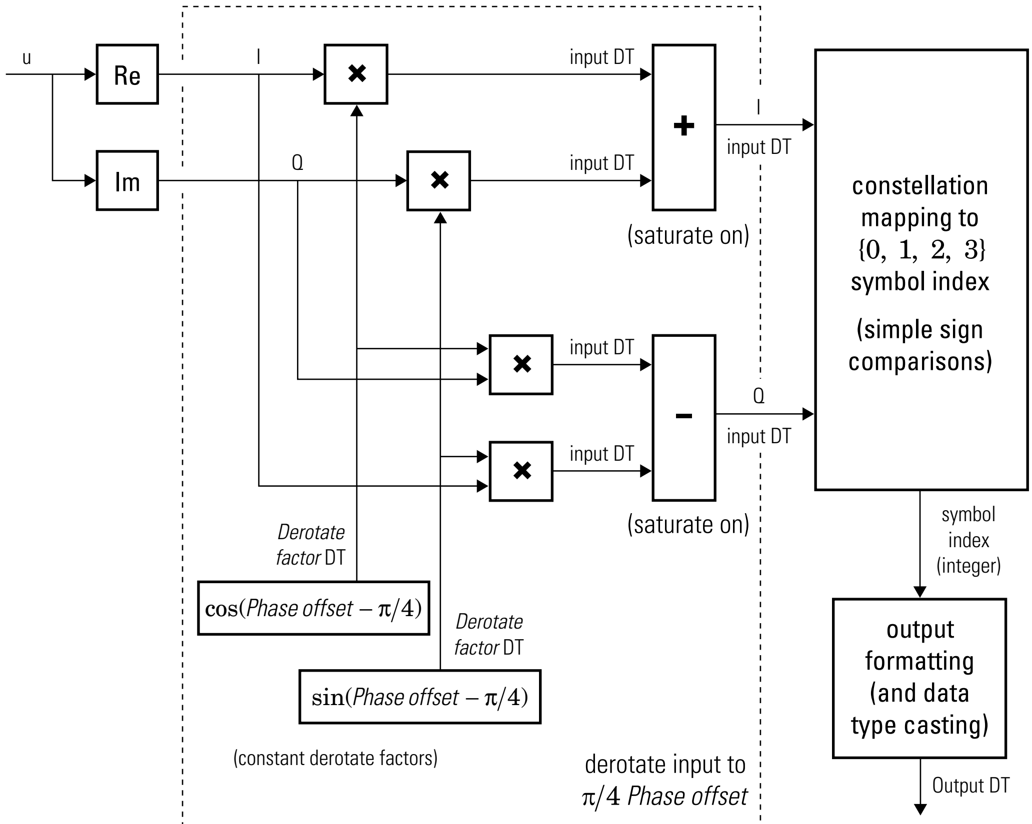
Algorithm



Hard-Decision QPSK Demodulator Signal Diagram for Trivial Phase Offset (odd multiple of $\frac{\pi}{4}$)



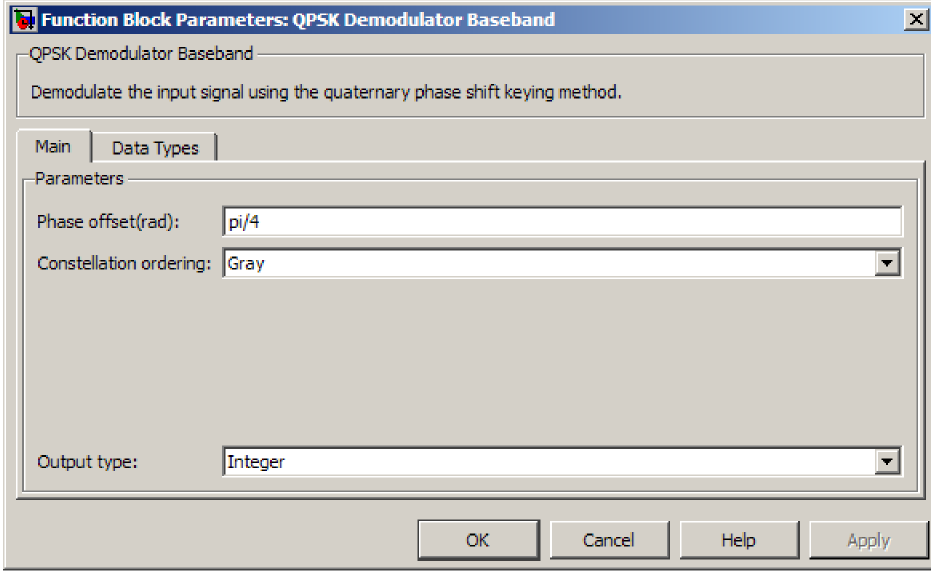
Hard-Decision QPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



Hard-Decision QPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

The exact LLR and approximate LLR cases (soft-decision) are described in “Exact LLR Algorithm” and “Approximate LLR Algorithm” in the *Communications System Toolbox User’s Guide*.

Dialog Box



Phase offset (rad)
The phase of the zeroth point of the signal constellation.

Constellation ordering
Determines how the block maps each integer to a pair of output bits.

Output type
Determines whether the output consists of integers or bits.

If the **Output type** parameter is set to `Integer` and **Constellation ordering** is set to `Binary`, then the block maps the point

$$\exp(j\theta + j\pi m/2)$$

to m, where θ is the **Phase offset** parameter and m is 0, 1, 2, or 3.

The reference page for the QPSK Modulator Baseband block shows the signal constellations for the cases when **Constellation ordering** is set to either `Binary` or `Gray`.

If the **Output type** is set to `Bit`, then the output contains pairs of binary values if **Decision type** is set to `Hard decision`. The most significant bit (i.e. the left-most bit in the vector), is the first bit the block outputs.

If the **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, then the output contains bitwise LLR or approximate LLR values, respectively.

Decision type

Specifies the use of hard decision, LLR, or approximate LLR during demodulation. This parameter appears when you select `Bit` from the **Output type** drop-down list. The output values for Log-likelihood ratio and Approximate log-likelihood ratio decision types are of the same data type as the input values. For integer output, the block always performs Hard decision demodulation.

See "Exact LLR Algorithm" and "Approximate LLR Algorithm" in the *Communications System Toolbox User's Guide* for algorithm details.

Noise variance source

This field appears when `Approximate log-likelihood ratio` or `Log-likelihood ratio` is selected for **Decision type**.

When set to `Dialog`, the noise variance can be specified in the **Noise variance** field. When set to `Port`, a port appears on the block through which the noise variance can be input.

Noise variance

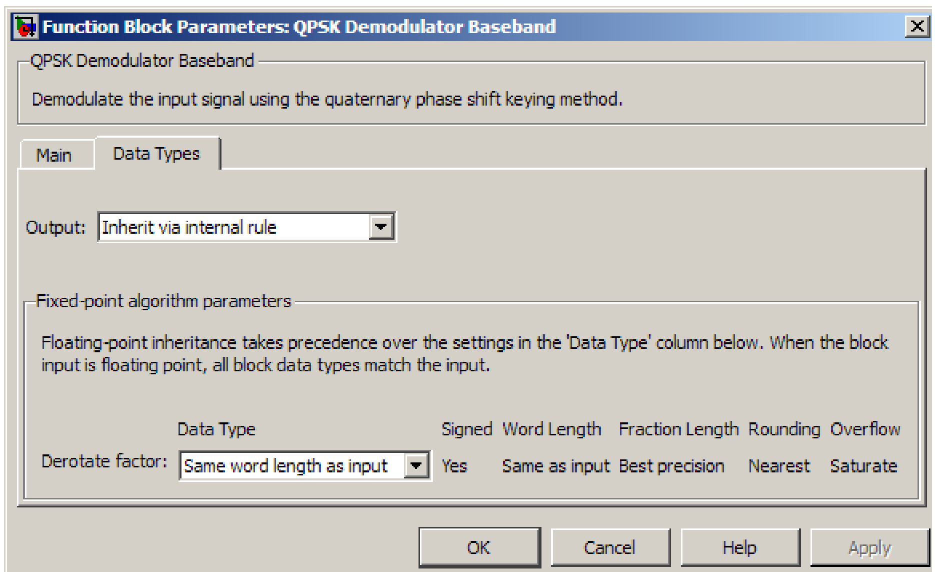
This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

The LLR algorithm involves computing exponentials of very large or very small numbers using finite precision arithmetic and would yield:

- `Inf` to `-Inf` if **Noise variance** is very high
- `NaN` if **Noise variance** and signal power are both very small

In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.



Data Types Pane for Hard-Decision

Output

For bit outputs, when **Decision type** is set to `Hard decision`, the output data type can be set to `'Inherit via internal rule'`, `'Smallest unsigned integer'`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `boolean`.

For integer outputs, the output data type can be set to `'Inherit via internal rule'`, `'Smallest unsigned integer'`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

When this parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is a floating-point type (`single` or `double`). If the input data type is fixed-point, the output data type will work as if this parameter is set to `'Smallest unsigned integer'`.

When this parameter is set to `'Smallest unsigned integer'`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model.

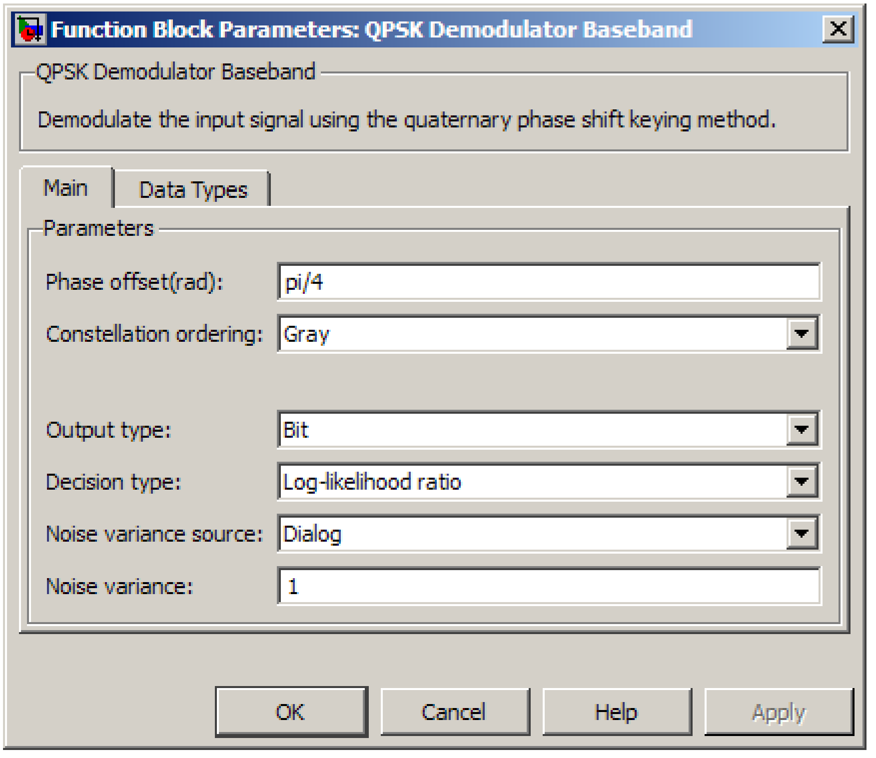
If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, and **Output type** is `Bit`, the output data type is the ideal minimum one-bit size, i.e., `ufix(1)`. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding to the size of a char (e.g., `uint8`).

If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, and **Output type** is `Integer`, the output data type is the ideal minimum two-bit size, i.e., `ufix(2)`. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit two bits, usually corresponding to the size of a char (e.g., `uint8`).

Derotate factor

This parameter only applies when the input is fixed-point and **Phase offset** is not an even multiple of $\frac{\pi}{4}$.

You can select `Same word length as input` or `Specify word length`, in which case you define the word length using an input field.



Data Types Pane for Soft-Decision

For bit outputs, when **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the output data type is inherited from the input (e.g., if the input is of data type `double`, the output is also of data type `double`).

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed-point when: - **Output type** is `Integer` - **Output type** is `Bit` and **Decision type** is `Hard-decision`
Var	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point • Boolean when **Output type** is `Bit` and **Decision type** is `Hard-decision` • 8-, 16-, 32- bit signed integers • 8-, 16-, 32- bit unsigned integers • ufix(1) in ASIC/FPGA when **Output type** is `Bit` • ufix(2) in ASIC/FPGA when **Output type** is `Integer`

Pair Block

QPSK Modulator Baseband

See Also

M-PSK Demodulator Baseband, BPSK Demodulator Baseband, DQPSK Demodulator Baseband

QPSK Modulator Baseband

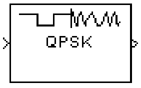
Purpose

Modulate using quaternary phase shift keying method

Library

PM in Digital Baseband sublibrary of Modulation

Description



The QPSK Modulator Baseband block modulates using the quaternary phase shift keying method. The output is a baseband representation of the modulated signal.

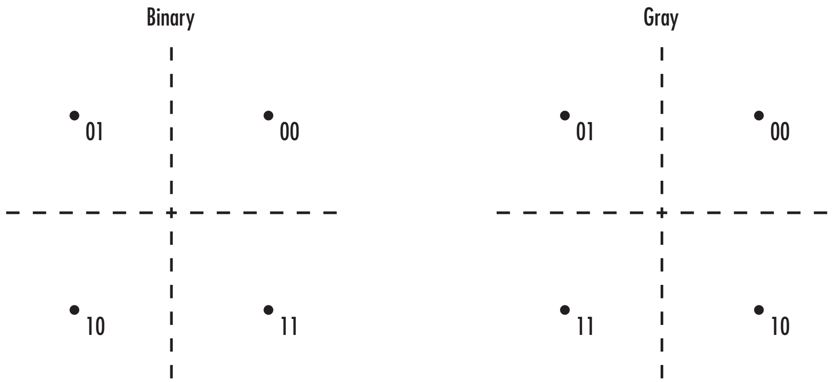
Integer-Valued Signals and Binary-Valued Signals

If you set the **Input type** parameter to `Integer`, then valid input values are 0, 1, 2, and 3. When you set **Constellation ordering** to `Binary` for input *m* the output symbol is

$$\exp(j\theta + j\pi m/2)$$

where θ represents the **Phase offset** parameter (see the following figure for Gray constellation ordering). In this case, the block accepts a scalar or column vector signal.

If you set the **Input type** parameter to `Bit`, then the input contains pairs of binary values. For this configuration, the block accepts column vectors with even lengths. When you set the **Phase offset** parameter to $\frac{\Pi}{4}$, then the block uses one of the signal constellations in the following figure, depending on whether you set the **Constellation ordering** parameter to `Binary` or `Gray`.

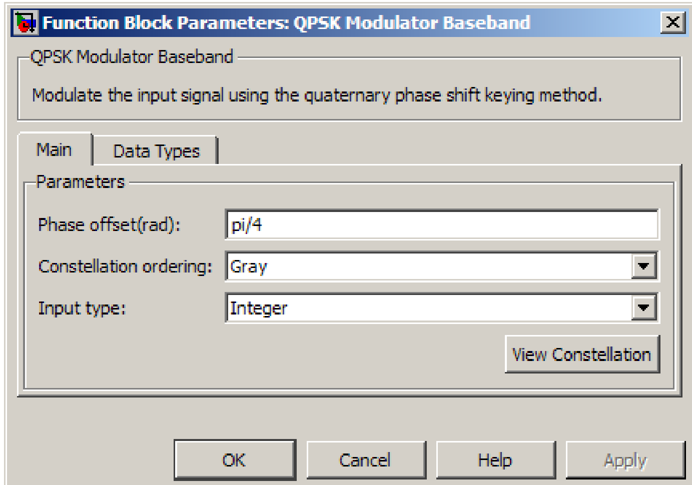


In the previous figure, the most significant bit (i.e. the left-most bit), is the first bit input to the block. For additional information about Gray mapping, see the M-PSK Modulator Baseband help page.

Constellation Visualization

The QPSK Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. For more information, see the Constellation Visualization section of the *Communications System Toolbox User's Guide*.

Dialog Box



Phase offset (rad)
: The phase of the zeroth point of the signal constellation.

Constellation ordering
: Determines how the block maps each pair of input bits or input integers to constellation symbols.

Input type
: Indicates whether the input consists of integers or pairs of bits.

Output data type
: The output data type can be set to `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

  Setting this parameter to `Fixed-point` or `User-defined` enables fields in which you can further specify details. Setting this parameter to `Inherit via back propagation`, sets the output data type and scaling to match the following block.

Output 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.

Set output fraction length to

Specify the scaling of the fixed-point output by either of the following 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 **Output 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.

User-defined data type

Specify any signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Output 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 output fraction length to** parameter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean when **Input type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) when **Input type** is `Bit` • ufix(2) when **Input type** is `Integer`
Output	• Double-precision floating point • Single-precision floating point • Signed fixed point

Pair Block

QPSK Demodulator Baseband

See Also

M-PSK Modulator Baseband, BPSK Modulator Baseband, DQPSK Modulator Baseband

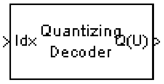
Purpose

Decode quantization index according to codebook

Library

Source Coding

Description

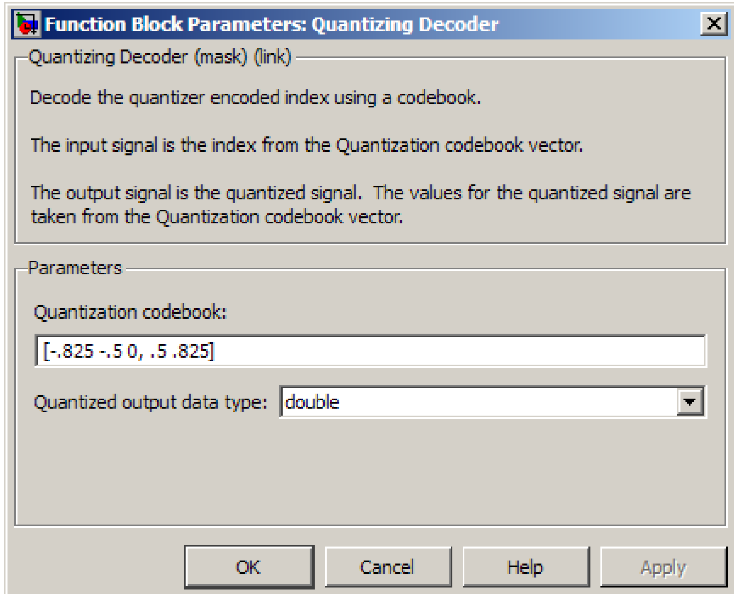


The Quantizing Decoder block converts quantization indices to the corresponding codebook values. The **Quantization codebook** parameter, a vector of length N, prescribes the possible output values. If the input is an integer k between 0 and N-1, then the output is the (k+1)st element of **Quantization codebook**.

The input must be a discrete-time signal. This block processes each vector element independently. For information about the data types each block port supports, see the “Supported Data Type” on page 2-755 table on this page.

Note The Quantizing Encoder block also uses a **Quantization codebook** parameter. The first output of that block corresponds to the input of Quantizing Decoder, while the second output of that block corresponds to the output of Quantizing Decoder.

Dialog Box



Quantization codebook

A real vector that prescribes the output value corresponding to each nonnegative integer of the input.

Quantized output data type

Select the output data type.

Supported Data Type

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

Pair Block

Quantizing Encoder

See Also

Scalar Quantizer (Obsolete) (DSP System Toolbox documentation)

Quantizing Encoder

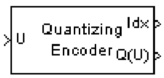
Purpose

Quantize signal using partition and codebook

Library

Source Coding

Description



The Quantizing Encoder block quantizes the input signal according to the **Partition** vector and encodes the input signal according to the **Codebook** vector. This block processes each vector element independently. The input must be a discrete-time signal. This block processes each vector element independently. For information about the data types each block port supports, see the "Supported Data Type" on page 2-758 table on this page.

The first output is the quantization index. The second output is the quantized signal. The values for the quantized signal are taken from the **Codebook** vector.

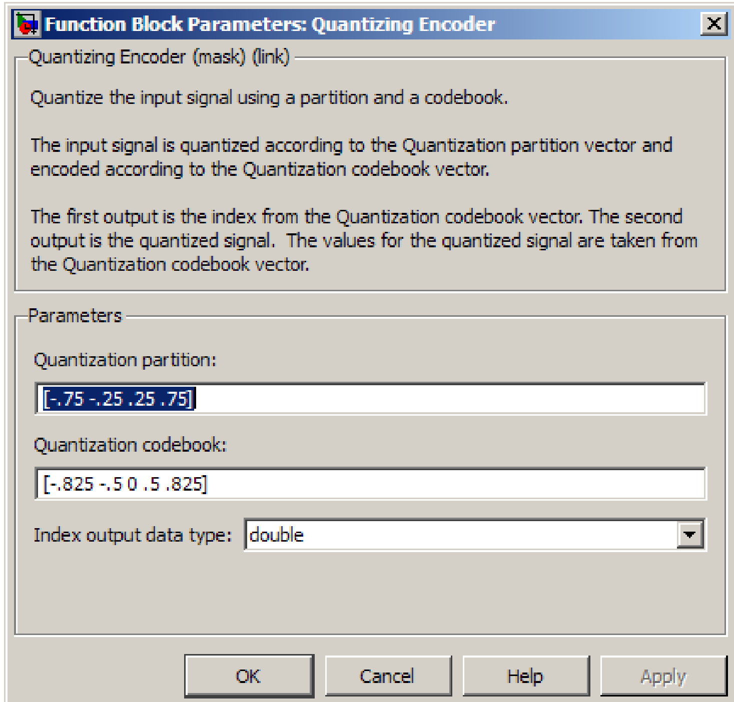
The **Quantization partition** parameter, P, is a real vector of length n whose entries are in strictly ascending order. The quantization index (second output signal value) corresponding to an input value of x is

- 0 if x P(1)
- m if P(m) < x P(m+1)
- n if P(n) < x

The **Quantization codebook** parameter, whose length is n+1, prescribes a value for each partition in the quantization. The first element of **Quantization codebook** is the value for the interval between negative infinity and the first element of P. The second output signal from this block contains the quantization of the input signal based on the quantization indices and prescribed values.

You can use the function `lloyds` in Communications System Toolbox with a representative sample of your data as training data, to obtain appropriate partition and codebook parameters.

Dialog Box



Quantization partition
: The vector of endpoints of the partition intervals.

Quantization codebook
: The vector of output values assigned to each partition.

Index output data type
: Select the output data type.

Supported Data Type

Port	Supported Data Types
U	• Double-precision floating point • Single-precision floating point • Signed fixed-point
Idx	• Double-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Q(U)	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

Quantizing Decoder

See Also

Scalar Quantizer (Obsolete) (DSP System Toolbox documentation), `lloyds` (Communications System Toolbox documentation)

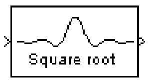
Purpose

Filter input signal, possibly downsampling, using raised cosine FIR filter

Library

Comm Filters

Description



The Raised Cosine Receive Filter block filters the input signal using a normal raised cosine FIR filter or a square root raised cosine FIR filter. It also downsamples the filtered signal if you set the **Output mode** parameter to `Downsampling`. The FIR Decimation block implements this functionality. The Raised Cosine Receive Filter block's icon shows the filter's impulse response.

Characteristics of the Filter

Characteristics of the raised cosine filter are the same as in the Raised Cosine Transmit Filter block, except that the length of the filter's input response has a slightly different expression: 2 * N * **Group delay** + 1, where N is the value of the **Input samples per symbol** parameter (not the **Upsampling factor** parameter, as in the case of the Raised Cosine Transmit Filter block).

If the **Filter gain** parameter is chosen to be `User-specified`, then the passband gain of the filter is:

- $20\log_{10}(\text{Input samples per symbol}(N)\times\text{Linear amplitude filter gain})$ for a normal filter.
- $20\log_{10}(\text{Input samples per symbol}(N)\times\text{Linear amplitude filter gain})$ for a square root filter.

Downsampling the Filtered Signal

To have the block downsample the filtered signal, set the **Output mode** parameter to `Downsampling`. By default, downsampling is on. If L is the **Downsampling factor** parameter value, then the block retains 1/L of the samples, choosing them as follows:

- If the **Sample offset** parameter is zero, then the block selects the samples of the filtered signal indexed by 1, L+1, 2*L+1, 3*L+1, etc.

- If the **Sample offset** parameter is a positive integer less than L, then the block initially discards that number of samples from the filtered signal and downsamples the remaining data as in the previous case.

To preserve the entire filtered signal and avoid downsampling, set **Output mode** to `None`. This setting is appropriate, for example, when the output from the filter block forms the input to a timing phase recovery block such as Squaring Timing Recovery. The timing phase recovery block performs the downsampling in that case.

Input Signals and Output Signals

This block accepts a column vector or matrix input signal. For information about the data types each block port supports, see the "Supported Data Type" on page 2-768 table on this page.

If you set **Output mode** to `None`, then the input and output signals share the same sampling mode, sample time, and vector length.

If you set **Output mode** to `Downsampling` and **Downsampling factor** is L, then L and the input sampling mode determine characteristics of the output signal:

Single-Rate Processing

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 genereate 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 $1/L$ times that of the input ($M_o = M_i/L$), In this mode, the input frame size, M_i, must be a multiple of L.

Multirate Processing

When you set the **Rate options** parameter to `Allow multirate processing`, the input and output of the block are the same size, but the sample rate of the output is K times slower than that of the input. When the block is in multirate processing mode, you must also specify a value for the **Input processing** parameter:

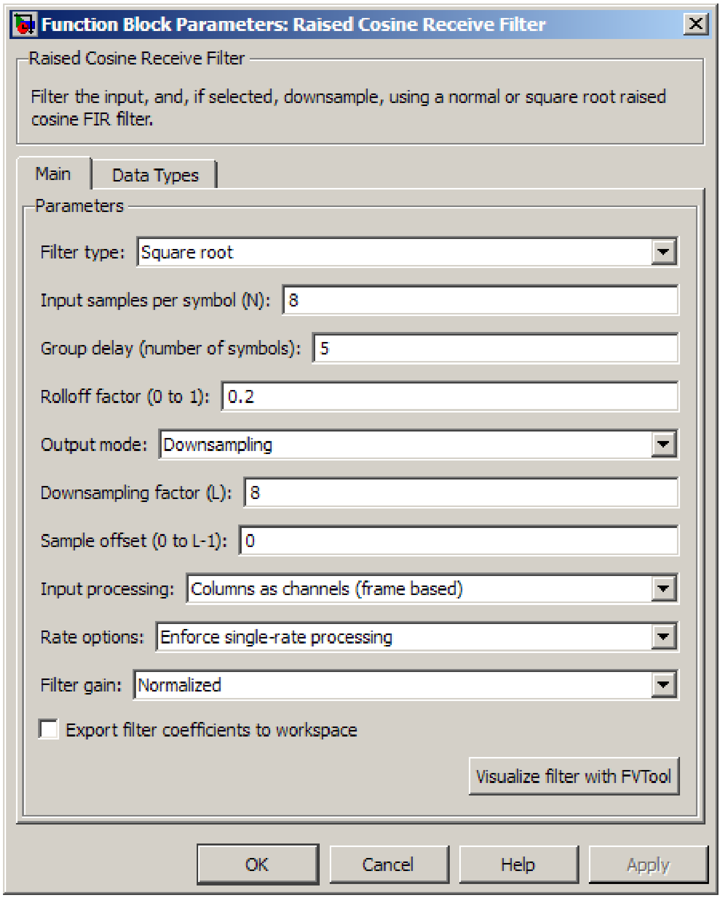
- When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-N matrix input as $M*N$ independent channels, and processes each channel over time. The output sample period (T_{so}) is L times longer than the input sample period ($T_{so} = L*T_{si}$), and the input and output sizes are identical.
- When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats an M_i-by-N matrix input as N independent channels. The block processes 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}) L times longer than the input frame period ($T_{fo} = L*T_{fi}$).

Exporting Filter Coefficients to the MATLAB Workspace

To examine or manipulate the coefficients of the filter that this block designs, select **Export filter coefficients to workspace**. Then set the **Coefficient variable name** parameter to the name of a variable that you want the block to create in the MATLAB workspace. Running the simulation causes the block to create the variable, overwriting any previous contents in case the variable already exists.

Latency

For information pertaining to the latency of the block, see details in FIR Decimation.



Dialog Box

Filter type

The type of raised cosine filter: `Square root` or `Normal`.

Input samples per symbol

An integer greater than 1 representing the number of samples per symbol in the input signal.

Group delay

A positive integer that represents the number of symbol periods between the start of the filter response and its peak. The default selection for this parameter is `5`.

Rolloff factor

The rolloff factor for the filter, a real number between 0 and 1.

Input processing

Specify how the block processes the input signal. 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 *Communications System Toolbox Release Notes* for more information.

Rate options

Specify the method by which the block should filter and downsample the input signal. 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 processes the signal by decreasing 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 processes the signal such that the output sample rate is L times slower than the input sample rate.

Output mode
: Determines whether or not the block downsamples the signal after filtering. Choices are `Downsampling` and `None`.

Downsampling factor
: The factor by which the block downsamples the signal after filtering. This field appears only if **Output mode** is set to `Downsampling`.

Sample offset
: The number of filtered samples the block discards before downsampling. This field appears only if **Output mode** is set to `Downsampling`.

Filter gain
: Determines how the block scales the filter coefficients. Choices are `Normalized` and `User-specified`.

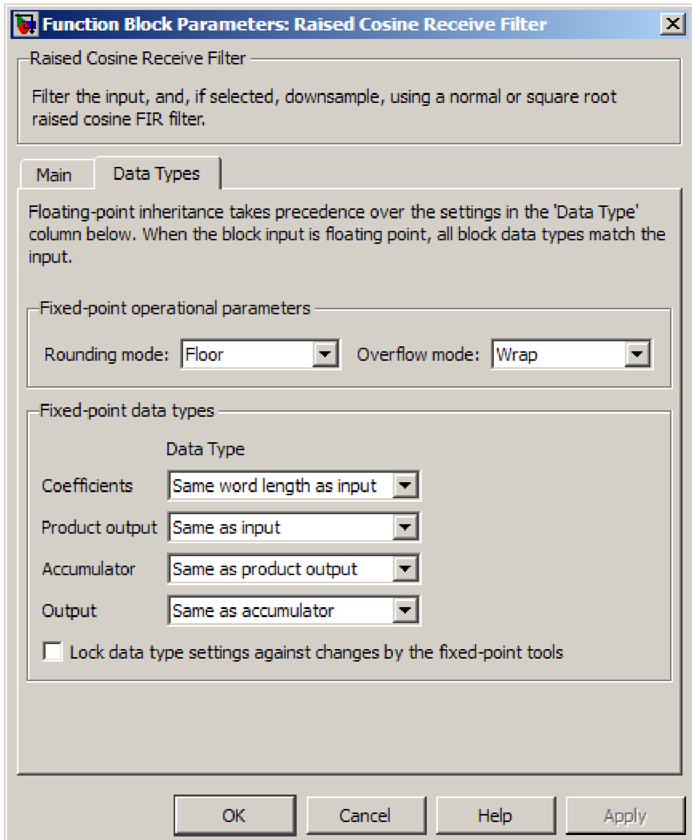
Linear amplitude filter gain
: A positive scalar used to scale the filter coefficients. This field appears only if **Filter gain** is set to `User-specified`.

Export filter coefficients to workspace
: Select this check box to create a variable in the MATLAB workspace that contains the filter coefficients.

Coefficient variable name
: The name of the variable to create in the MATLAB workspace. This field appears only if **Export filter coefficients to workspace** is selected.

Visualize filter with FVTool
: If you click this button, then MATLAB launches the Filter Visualization Tool, `fvtool`, to analyze the raised cosine filter whenever you apply any changes to the block's parameters. If you launch `fvtool` for the filter, and subsequently change parameters in the mask, `fvtool` will not update. You will need to launch a

new `fvtool` in order to see the new filter characteristics. Also note that if you have launched `fvtool`, then it will remain open even after the model is closed.



Rounding mode

Select the rounding mode for fixed-point operations. The block uses the **Rounding mode** when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. The filter coefficients do not obey this parameter; they always round to `Nearest`. For more information, see Rounding Modes in the DSP System Toolbox documentation or "Rounding Mode: Simplest" in the Fixed-Point Designer documentation.

Overflow mode

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

Coefficients

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator).

See the Coefficients section of the FIR Decimation help page and "Filter Structure Diagrams" in *DSP System Toolbox Reference Guide* for illustrations depicting the use of the coefficient data types in this block:

See the Coefficients subsection of the Digital Filter help page for descriptions of parameter settings.

- 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 are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits. If applicable, you are able to enter separate fraction lengths for the numerator and denominator coefficients.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. If applicable, you are able to enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.
- The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See "Filter Structure Diagrams" and "Multiplication Data Types" in *DSP System Toolbox Reference Guide* 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 are able to enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See "Filter Structure Diagrams" 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 are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

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

Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling tool in the Fixed-Point Tool.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Signed fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block Raised Cosine Transmit Filter

See Also Gaussian Filter, `rcosine`, `rcosflt`

Raised Cosine Transmit Filter

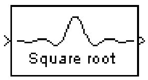
Purpose

Upsample and filter input signal using raised cosine FIR filter

Library

Comm Filters

Description



The Raised Cosine Transmit Filter block upsamples and filters the input signal using a normal raised cosine FIR filter or a square root raised cosine FIR filter. The block's icon shows the filter's impulse response.

Characteristics of the Filter

The **Filter type** parameter determines which type of filter the block uses; choices are `Normal` and `Square root`.

The impulse response of a normal raised cosine filter with rolloff factor R and symbol period T is

$$h(t) = \frac{\sin(\pi t / T)}{(\pi t / T)} \cdot \frac{\cos(\pi R t / T)}{(1 - 4R^2 t^2 / T^2)}$$

The impulse response of a square root raised cosine filter with rolloff factor R is

$$h(t) = 4R \frac{\cos\left((1+R)\pi t / T\right) + \dfrac{\sin\left((1-R)\pi t / T\right)}{(4Rt/T)}}{\pi\sqrt{T}\left(1 - (4Rt/T)^2\right)}$$

The impulse response of a square root raised cosine filter convolved with itself is approximately equal to the impulse response of a normal raised cosine filter.

The **Group delay** parameter is the number of symbol periods between the start of the filter's response and the peak of the filter's response. The group delay and the upsampling factor, N, determine the length of the filter's impulse response, which is 2 * N * **Group delay** + 1.

The **Rolloff factor** parameter is the filter's rolloff factor. It must be a real number between 0 and 1. The rolloff factor determines the excess bandwidth of the filter. For example, a rolloff factor of .5 means that the bandwidth of the filter is 1.5 times the input sampling frequency.

The **Filter gain** parameter indicates how the block normalizes the filter coefficients. If you choose `Normalized`, then the block uses an automatic scaling:

- If **Filter type** is `Normal`, then the block normalizes the filter coefficients so that the peak coefficient equals 1.
- If **Filter type** is `Square root`, then the block normalizes the filter coefficients so that the convolution of the filter with itself produces a normal raised cosine filter whose peak coefficient equals 1.

As a result, the output is scaled by $\sqrt{N}$. If the output of this block feeds the input to the AWGN Channel block, specify the AWGN signal power parameter to be 1/N.

If the **Filter gain** parameter is chosen to be `User-specified`, then the passband gain of the filter is:

- $20\log_{10}(\text{Upsampling factor}(N)\times\text{Linear amplitude filter gain})$ for a normal filter.
- $20\log_{10}(\sqrt{\text{Upsampling factor}(N)}\times\text{Linear amplitude filter gain})$ for a square root filter.

Input Signals and Output Signals

The input must be a discrete-time signal. This block accepts a column vector or matrix input signal. For information about the data types each block port supports, see the "Supported Data Type" on page 2-779 table on this page.

The **Rate options** method and the value of the **Upsampling factor (N)** parameter determine the characteristics of the output signal:

Single-Rate Processing

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 generate 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 N times larger than that of the input ($M_o = M_i*N$).

Multirate Processing

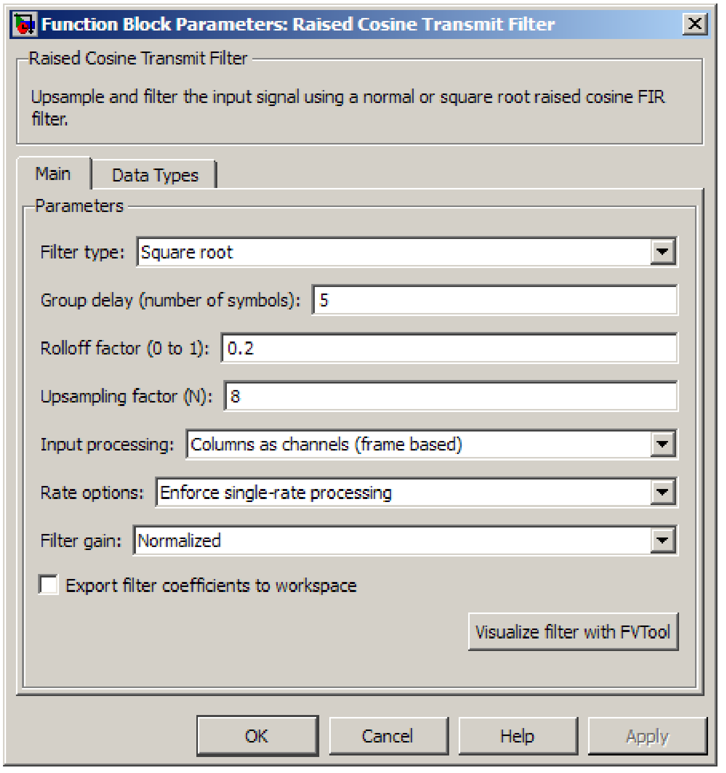
When you set the **Rate options** parameter to `Allow multirate processing`, the input and output of the block are the same size. However, the sample rate of the output is N times faster than that of the input (i.e. the output sample time is 1/N times the input sample time). When the block is in multirate processing mode, you must also specify a value for the **Input processing** parameter:

- When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-L matrix input as $M*L$ independent channels, and processes each channel over time. The output sample period (T_{so}) is N times shorter than the input sample period ($T_{so} = T_{si}/N$), while the input and output sizes remain identical.
- When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats an M_i-by-L matrix input as L independent channels. The block processes 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}) N times shorter than the input frame period ($T_{fo} = T_{fi}/N$).

Exporting Filter Coefficients to the MATLAB Workspace

To examine or manipulate the coefficients of the filter that this block designs, select **Export filter coefficients to workspace**. Then set the **Coefficient variable name** parameter to the name of a variable that you want the block to create in the MATLAB workspace. Running the simulation causes the block to create the variable, overwriting any previous contents in case the variable already exists.

Dialog Box



Filter type

The type of raised cosine filter: `Square root` or `Normal`.

Group delay

A positive integer that represents the number of symbol periods between the start of the filter response and its peak. The default selection for this parameter is `5`.

Rolloff factor

The rolloff factor for the filter, a real number between 0 and 1.

Upsampling factor

An integer greater than 1 representing the number of samples per symbol in the filtered output signal.

Input processing

Specify how the block processes the input signal. 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 *Communications System Toolbox Release Notes* for more information.

Rate options

Specify the method by which the block should upsample and filter the input signal. 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 processes the signal by increasing the output frame size by a factor of N. 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 processes the signal such that the output sample rate is N times faster than the input sample rate.

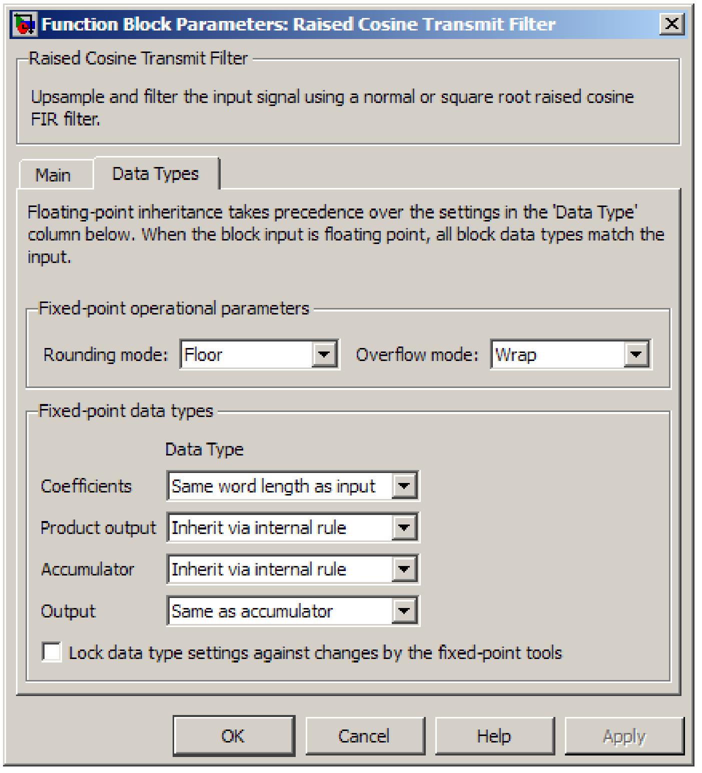
Filter gain
Determines how the block scales the filter coefficients. Choices are `Normalized` and `User-specified`.

Linear amplitude filter gain
A positive scalar used to scale the filter coefficients. This field appears only if **Filter gain** is set to `User-specified`.

Export filter coefficients to workspace
Select this check box to create a variable in the MATLAB workspace that contains the filter coefficients.

Coefficient variable name
The name of the variable to create in the MATLAB workspace. This field appears only if **Export filter coefficients to workspace** is selected.

Visualize filter with FVTool
If you click this button, then MATLAB launches the Filter Visualization Tool, `fvtool`, to analyze the raised cosine filter whenever you apply any changes to the block's parameters. If you launch `fvtool` for the filter, and subsequently change parameters in the mask, `fvtool` will not update. You will need to launch a new `fvtool` in order to see the new filter characteristics. Also note that if you have launched `fvtool`, then it will remain open even after the model is closed.



Rounding mode

Select the rounding mode for fixed-point operations. The block uses the **Rounding mode** when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. The filter coefficients do not obey this parameter; they always round to `Nearest`. For more information, see Rounding Modes in the DSP System Toolbox documentation or "Rounding Mode: Simplest" in the Fixed-Point Designer documentation.

Overflow mode

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

Coefficients

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). See “Filter Structure Diagrams” in *DSP System Toolbox Reference Guide* 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 are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits. If applicable, you are able to enter separate fraction lengths for the numerator and denominator coefficients.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. If applicable, you are able to enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.
- The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See “Filter Structure Diagrams” and “Multiplication Data Types” in *DSP System Toolbox Reference Guide* 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 are able to enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See “Filter Structure Diagrams” 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 are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

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

Lock scaling against changes by the autoscaling tool

Select this check box to prevent any fixed-point scaling you specify for this block mask from being overridden by the autoscaling tool in the Fixed-Point Tool.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Signed fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

Raised Cosine Receive Filter

See Also

Gaussian Filter, `rcosine`, `rcosflt`

Random Deinterleaver

Purpose

Restore ordering of input symbols using random permutation

Library

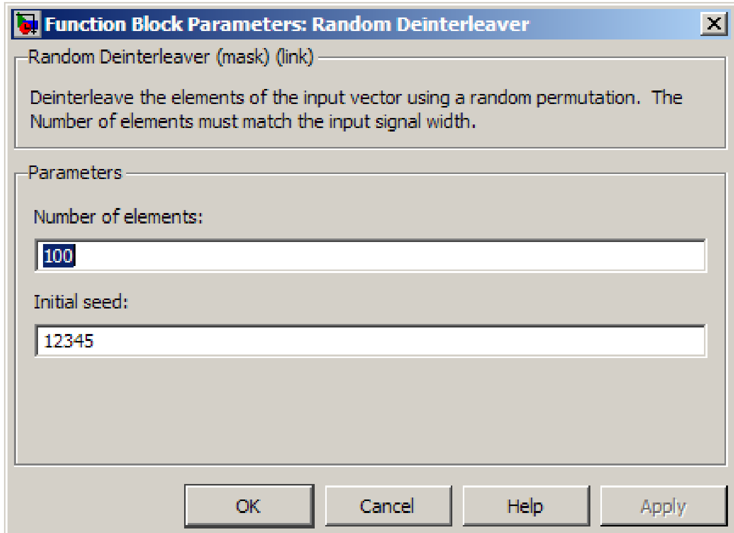
Block sublibrary of Interleaving

Description

The Random Deinterleaver block rearranges the elements of its input vector using a random permutation. The **Initial seed** parameter initializes the random number generator that the block uses to determine the permutation. If this block and the Random Interleaver block have the same value for **Initial seed**, then the two blocks are inverses of each other.

This block accepts a column vector input signal. The **Number of elements** parameter indicates how many numbers are in the input vector.

The block accepts the following data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.



Dialog Box

Number of elements

The number of elements in the input vector.

Initial seed

The initial seed value for the random number generator.

Pair Block

Random Interleaver

See Also

General Block Deinterleaver

Random Integer Generator

Purpose

Generate integers randomly distributed in range [0, M-1]

Library

Random Data Sources sublibrary of Comm Sources

Description

The Random Integer Generator block generates uniformly distributed random integers in the range [0, *M*-1], where *M* is the **M-ary number** defined in the dialog box.

The **M-ary number** can be either a scalar or a vector. If it is a scalar, then all output random variables are independent and identically distributed (i.i.d.). If the **M-ary number** is a vector, then its length must equal the length of the **Initial seed**; in this case each output has its own output range.

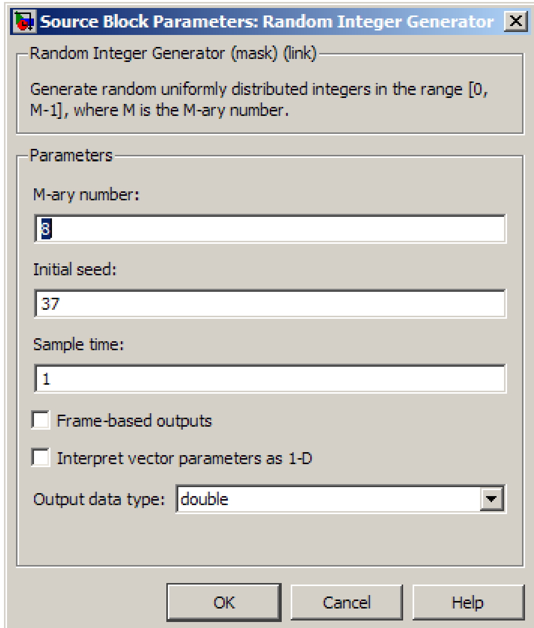
If the **Initial seed** parameter is a constant, then the resulting noise is repeatable.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples per frame**, and **Interpret vector parameters as 1-D** parameters. See "Sources and Sinks" in *Communications System ToolboxUser's Guide* for more details.

The number of elements in the **Initial seed** parameter becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. Also, the shape (row or column) of the **Initial seed** parameter becomes the shape of a sample-based two-dimensional output signal.

Dialog Box



M-ary number
: The positive integer, or vector of positive integers, that indicates the range of output values.

Initial seed
: The initial seed value for the random number generator. The vector length of the seed determines the length of the output vector.

Sample time
: The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs
: Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame
The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D
If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type
The output type of the block can be specified as a `boolean`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, or `double`. By default, the block sets this to `double`. Single outputs may lead to different results when compared with double outputs for the same set of parameters. For Boolean typed outputs, the **M-ary number** must be `2`.

See Also

`randint` (Communications Toolbox)

Purpose

Reorder input symbols using random permutation

Library

Block sublibrary of Interleaving

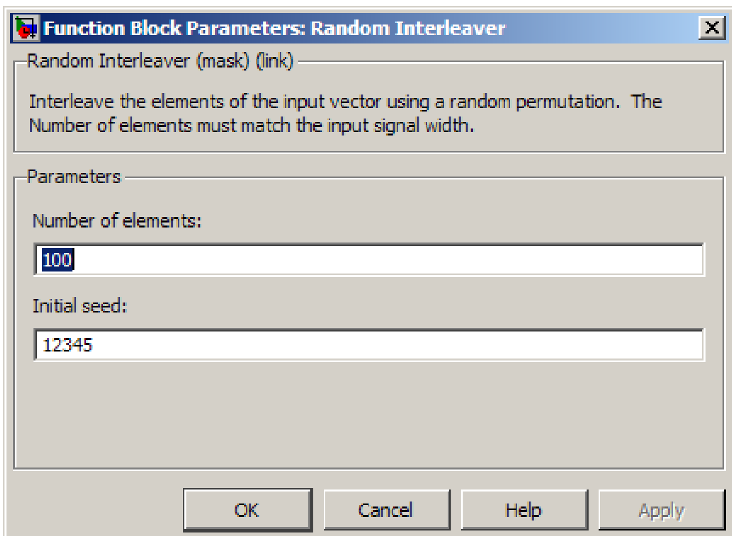
Description

The Random Interleaver block rearranges the elements of its input vector using a random permutation. This block accepts a column vector input signal. The **Number of elements** parameter indicates how many numbers are in the input vector.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

The **Initial seed** parameter initializes the random number generator that the block uses to determine the permutation. The block is predictable for a given seed, but different seeds produce different permutations.

Dialog Box



Number of elements

The number of elements in the input vector.

Initial seed
The initial seed value for the random number generator.

Pair Block

Random Deinterleaver

See Also

General Block Interleaver

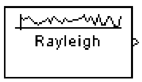
Purpose

Generate Rayleigh distributed noise

Library

Noise Generators sublibrary of Comm Sources

Description



The Rayleigh Noise Generator block generates Rayleigh distributed noise. The Rayleigh probability density function is given by

$$f(x) = \begin{cases} \dfrac{x}{\sigma^2} \exp\left(-\dfrac{x^2}{2\sigma^2}\right) & x \ge 0 \\ 0 & x < 0 \end{cases}$$

where σ^2 is known as the *fading envelope* of the Rayleigh distribution.

The block requires you to specify the **Initial seed** for the random number generator. If it is a constant, then the resulting noise is repeatable. The **sigma** parameter can be either a vector of the same length as the **Initial seed**, or a scalar. When **sigma** is a scalar, every element of the output signal shares that same value.

Initial Seed

The **Initial seed** parameter initializes the random number generator that the Rayleigh Noise Generator block uses to add noise to the input signal. For best results, the **Initial seed** should be a prime number greater than 30. Also, if there are other blocks in a model that have an **Initial seed** parameter, you should choose different initial seeds for all such blocks.

You can choose seeds for the Rayleigh Noise Generator block using the Communications System Toolbox randseed function. At the MATLAB prompt, enter

```
randseed
```

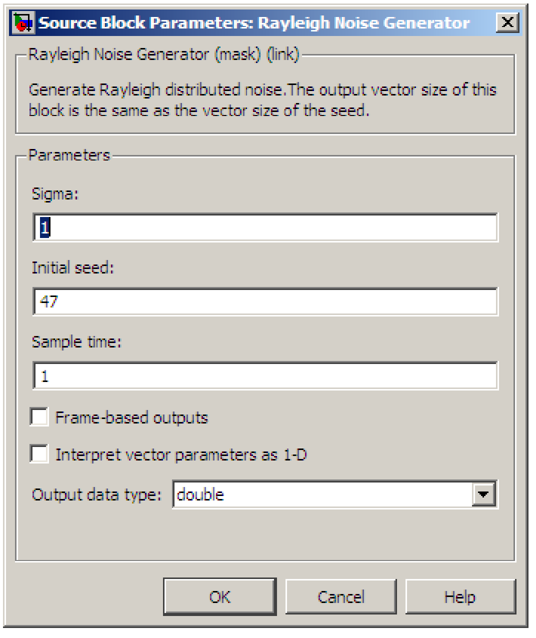
This returns a random prime number greater than 30. Entering `randseed` again produces a different prime number. If you supply an integer argument, `randseed` always returns the same prime for that integer. For example, `randseed(5)` always returns the same answer.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples per frame**, and **Interpret vector parameters as 1-D** parameters. See “Sources and Sinks” in the Control System Toolbox™ documentation for more details.

The number of elements in the **Initial seed** parameter becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. Also, the shape (row or column) of the **Initial seed** parameter becomes the shape of a sample-based two-dimensional output signal.

Dialog Box



Sigma

Specify σ as defined in the Rayleigh probability density function.

Initial seed
The initial seed value for the random number generator.

Sample time
The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs
Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame
The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D
If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type
The output can be set to `double` or `single` data types.

See Also

Multipath Rayleigh Fading Channel; `raylrnd` (Statistics Toolbox)

References

[1] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

Receiver Thermal Noise

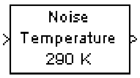
Purpose

Apply receiver thermal noise to complex baseband signal

Library

RF Impairments

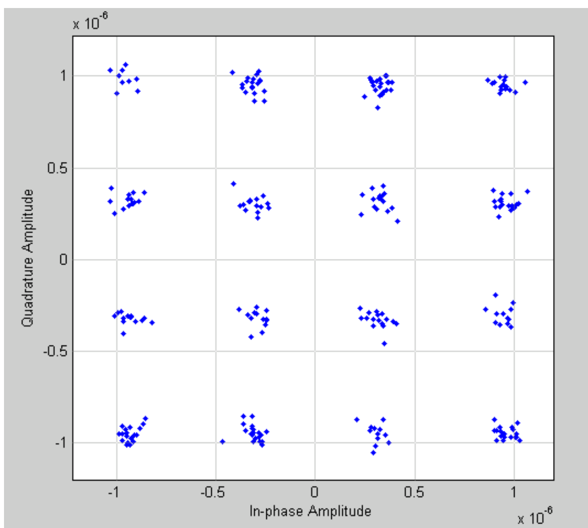
Description



The Receiver Thermal Noise block simulates the effects of thermal noise on a complex, baseband signal. You can specify the amount of thermal noise in three ways, according to which **Specification method** you select:

- `Noise temperature` specifies the noise in degrees kelvin.
- `Noise factor` specifies the noise as 1+(Noise temperature / 290).
- `Noise figure` specifies the noise as 10*log10(1+(Noise temperature / 290)). This is the decibel equivalent of Noise factor.

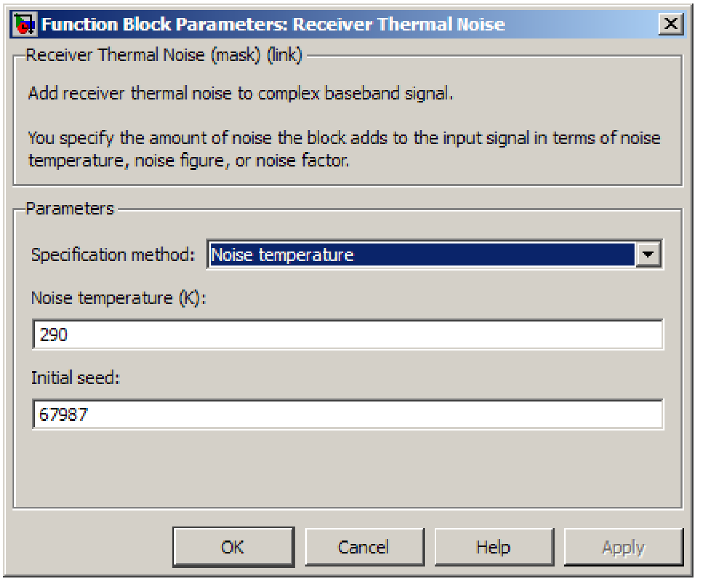
The following scatter plot shows the effect of the Receiver Thermal Noise block, with **Specification method** set to `Noise figure` and **Noise figure (dB)** set to `3.01`, on a signal modulated by 16-QAM.



This plot is generated by the model described in "Illustrate RF Impairments That Distort a Signal" with the following parameter settings:

- Rectangular QAM Modulator Baseband
  - **Normalization method** set to `Average Power`
  - **Average power (watts)** set to `1e-12`
- Receiver Thermal Noise
  - **Specification method** set to `Noise figure`
  - **Noise figure (dB)** set to `3.01`

Dialog Box



Specification method

The method by which you specify the amount of noise. The choices are `Noise temperature`, `Noise figure`, and `Noise factor`.

Noise temperature (K)
: Scalar specifying the amount of noise in degrees kelvin.

Noise figure
: Scalar specifying the amount of noise in decibels relative to a noise temperature of 290 degrees kelvin. A **Noise figure** setting of 0 dB indicates a noiseless system.

Noise factor
: Scalar specifying the amount of noise relative to a noise temperature of 290 degrees kelvin.

Initial seed
: The initial seed value for the random number generator that generates the noise.

See Also

Free Space Path Loss

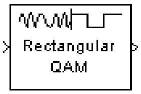
Purpose

Demodulate rectangular-QAM-modulated data

Library

AM, in Digital Baseband sublibrary of Modulation

Description



The Rectangular QAM Demodulator Baseband block demodulates a signal that was modulated using quadrature amplitude modulation with a constellation on a rectangular lattice.

Note All values of power assume a nominal impedance of 1 ohm.

The signal constellation has M points, where M is the **M-ary number** parameter. M must have the form 2^K for some positive integer K. The block scales the signal constellation based on how you set the **Normalization method** parameter. For details, see the reference page for the Rectangular QAM Modulator Baseband block.

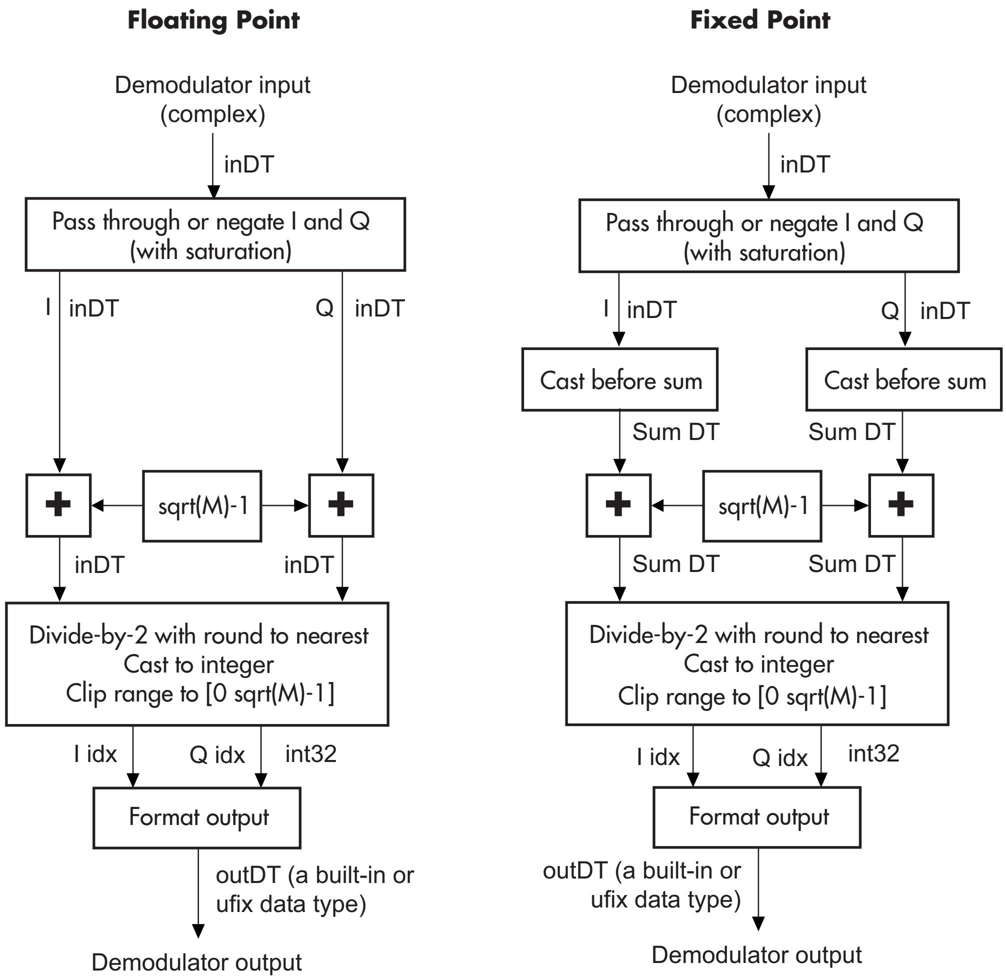
This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see the "Supported Data Types" on page 2-804 table on this page.

Hard Decision Algorithm

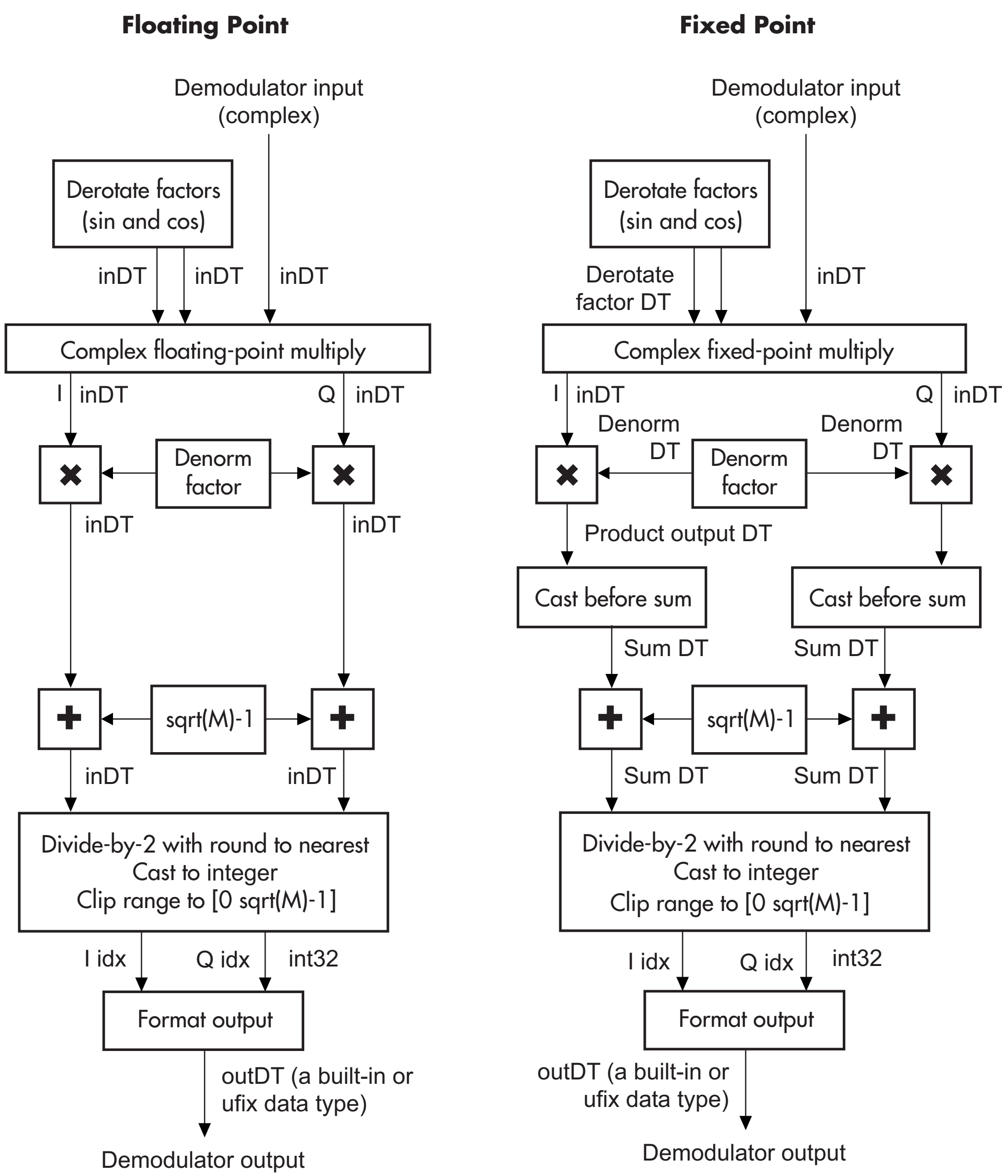
The demodulator algorithm maps received input signal constellation values to M-ary integer I and Q symbol indices between 0 and $\sqrt{M}-1$ and then maps these demodulated symbol indices to formatted output values.

The integer symbol index computation is performed by first derotating and scaling the complex input signal constellation (possibly with noise) by a derotate factor and denormalization factor, respectively. These factors are derived from the **Phase offset**, **Normalization method**, and related parameters. These derotated and denormalized values are added to $\sqrt{M}-1$ to translate them into an approximate range between 0 and $2\times(\sqrt{M}-1)$ (plus noise). The resulting values are then rescaled via a divide-by-two (or, equivalently, a right-shift by one bit for fixed-point operation) to obtain a range approximately between 0 and $\sqrt{M}-1$ (plus noise) for I and Q. The noisy index values are rounded to the nearest integer and clipped, via saturation, and mapped to integer symbol values in the range [0 M-1]. Finally, based on other block parameters, the integer index is mapped to a symbol value that is formatted and cast to the selected **Output data type**.

The following figures contains signal flow diagrams for floating-point and fixed-point algorithm operation. The floating-point diagrams apply when the input signal data type is `double` or `single`. The fixed-point diagrams apply when the input signal is a signed fixed-point data type. Note that the diagram is simplified when **Phase offset** is a multiple of $\frac{\pi}{2}$, and/or the derived denormalization factor is 1.

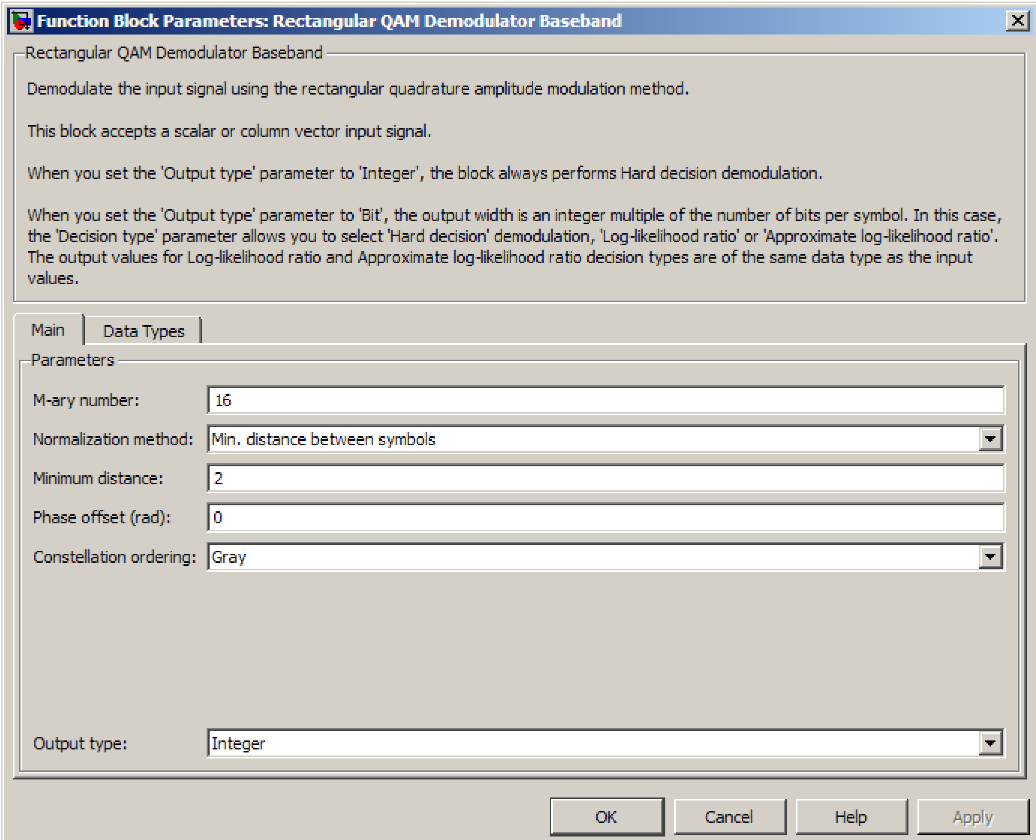


Signal-Flow Diagrams with Trivial Phase Offset and Denormalization Factor Equal to 1



Signal-Flow Diagrams with Nontrivial Phase Offset and Nonunity Denormalization Factor

Dialog Box



M-ary number

The number of points in the signal constellation. It must have the form 2^K for some positive integer K.

Normalization method

Determines how the block scales the signal constellation. Choices are `Min. distance between symbols`, `Average Power`, and `Peak Power`.

Minimum distance

This parameter appears when **Normalization method** is set to `Min. distance between symbols`.

The distance between two nearest constellation points.

Average power, referenced to 1 ohm (watts)

The average power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Average Power`.

Peak power, referenced to 1 ohm (watts)

The maximum power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Peak Power`.

Phase offset (rad)

The rotation of the signal constellation, in radians.

Constellation ordering

Determines how the block assigns binary words to points of the signal constellation. More details are on the reference page for the Rectangular QAM Modulator Baseband block.

Selecting `User-defined` displays the field **Constellation mapping**, allowing for user-specified mapping.

Constellation mapping

This parameter appears when `User-defined` is selected in the pull-down list **Constellation ordering**.

This is a row or column vector of size M and must have unique integer values in the range [0, M-1]. The values must be of data type `double`.

The first element of this vector corresponds to the top-leftmost point of the constellation, with subsequent elements running down column-wise, from left to right. The last element corresponds to the bottom-rightmost point.

Output type

Determines whether the block produces integers or binary representations of integers.

If set to `Integer`, the block produces integers.

If set to `Bit`, the block produces a group of K bits, called a *binary word*, for each symbol, when **Decision type** is set to `Hard decision`. If **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the block outputs bitwise LLR and approximate LLR, respectively.

Decision type

This parameter appears when `Bit` is selected in the pull-down list **Output type**.

Specifies the use of hard decision, LLR, or approximate LLR during demodulation. See “Exact LLR Algorithm” and “Approximate LLR Algorithm” in the *Communications System Toolbox User’s Guide* for algorithm details.

Noise variance source

This parameter appears when `Approximate log-likelihood ratio` or `Log-likelihood ratio` is selected for **Decision type**.

When set to `Dialog`, the noise variance can be specified in the **Noise variance** field. When set to `Port`, a port appears on the block through which the noise variance can be input.

Noise variance

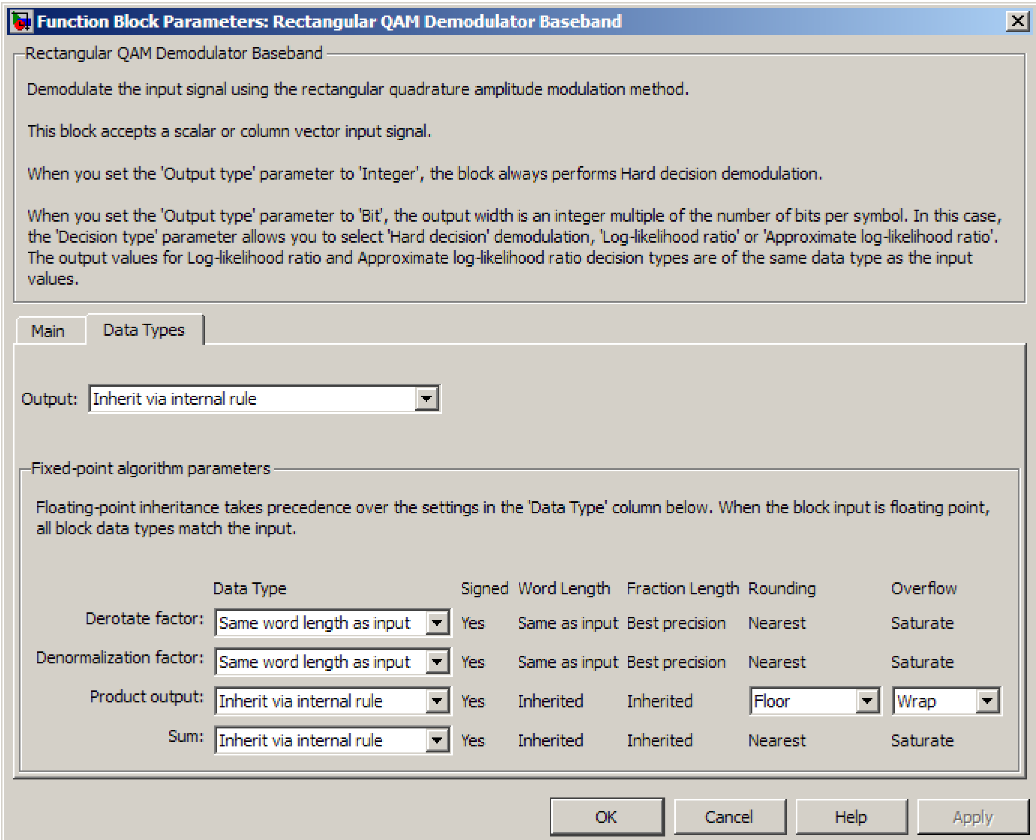
This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

The LLR algorithm involves computing exponentials of very large or very small numbers using finite precision arithmetic and would yield:

- `Inf` to `-Inf` if **Noise variance** is very high
- `NaN` if **Noise variance** and signal power are both very small

In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.



Output

When the parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is of type `single` or `double`. Otherwise, the output data type will be as if this parameter is set to `'Smallest unsigned integer'`.

When the parameter is set to 'Smallest unsigned integer', the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If ASIC/FPGA is selected in the **Hardware Implementation** pane, the output data type is the ideal minimum size, i.e., ufix(1) for bit outputs, and ufix(ceil(log2(M))) for integer outputs. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit the ideal minimum size, usually corresponding to the size of a char (e.g., uint8).

For integer outputs, this parameter can be set to Smallest unsigned integer, int8, uint8, int16, uint16, int32, uint32, single, and double. For bit outputs, the options are Smallest unsigned integer, int8, uint8, int16, uint16, int32, uint32, boolean, single, or double.

Derotate factor

This parameter only applies when the input is fixed-point and **Phase offset** is not a multiple of $\frac{\pi}{2}$.

This can be set to Same word length as input or Specify word length, in which case a field is enabled for user input.

Denormalization factor

This parameter only applies when the input is fixed-point and the derived denormalization factor is nonunity (not equal to 1). This scaling factor is derived from **Normalization method** and other parameter values in the block dialog.

This can be set to Same word length as input or Specify word length, in which case a field is enabled for user input. A best-precision fraction length is always used.

Product output

This parameter only applies when the input is a fixed-point signal and there is a nonunity (not equal to 1) denormalized factor. It can be set to `Inherit via internal rule` or `Specify word length`, which enables a field for user input.

Setting to `Inherit via internal rule` computes the full-precision product word length and fraction length. Internal Rule for Product Data Types in *DSP System Toolbox User's Guide* describes the full-precision **Product output** internal rule.

Setting to `Specify word length` allows you to define the word length. The block computes a best-precision fraction length based on the word length specified and the pre-computed worst-case (min/max) real world value **Product output** result. The worst-case **Product output** result is precomputed by multiplying the denormalized factor with the worst-case (min/max) input signal range, purely based on the input signal data type.

The block uses the **Rounding mode** when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. For more information, see "Rounding Modes" in the DSP System Toolbox documentation or "Rounding Mode: Simplest" in the Fixed-Point Designer documentation.

Sum

This parameter only applies when the input is a fixed-point signal. It can be set to `Inherit via internal rule`, `Same as product output`, or `Specify word length`, in which case a field is enabled for user input

Setting to `Inherit via internal rule` computes the full-precision sum word length and fraction length, based on the two inputs to the Sum in the fixed-point Hard Decision Algorithm signal flow diagram. The rule is the same as the fixed-point inherit rule of the internal **Accumulator data type** parameter in the Simulink Sum block.

Setting to `Specify word length` allows you to define the word length. A best precision fraction length is computed based on the word length specified in the pre-computed maximum range necessary for the demodulated algorithm to produce accurate results. The signed fixed-point data type that has the best precision fully contains the values in the range $2*(\sqrt{M}-1)$ for the specified word length.

Setting to `Same as product output` allows the Sum data type to be the same as the **Product output** data type (when **Product output** is used). If the **Product output** is not used, then this setting will be ignored and the `Inherit via internal rule` Sum setting will be used.

Supported Data Types

Port	Supported Data Types

Input	• Double-precision floating point
	• Single-precision floating point
	• Signed fixed–point when **M-ary number** is an even power of `2` and:
	  - **Output type** is `Integer`
	  - **Output type** is `Bit` and **Decision type** is `Hard-decision`

Var	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point • Boolean when **Output type** is `Bit` • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers • ufix(1) in ASIC/FPGA when **Output type** is `Bit` • $ufix(\log_2 M)$ in ASIC/FPGA when **Output type** is `Integer`

Pair Block

Rectangular QAM Modulator Baseband

See Also

General QAM Demodulator Baseband

References

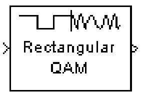
[1] Smith, Joel G., "Odd-Bit Quadrature Amplitude-Shift Keying," *IEEE Transactions on Communications*, Vol. COM-23, March 1975, 385–389.

Rectangular QAM Modulator Baseband

Purpose Modulate using rectangular quadrature amplitude modulation

Library AM, in Digital Baseband sublibrary of Modulation

Description The Rectangular QAM Modulator Baseband block modulates using M-ary quadrature amplitude modulation with a constellation on a rectangular lattice. The output is a baseband representation of the modulated signal. This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see "Supported Data Types" on page 2-812.



Note All values of power assume a nominal impedance of 1 ohm.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, the block accepts integer values between `0` and M-`1`. M represents the **M-ary number** block parameter.

When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

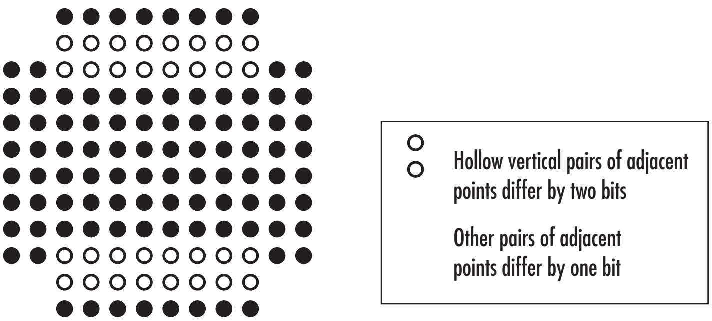
where

K represents the number of bits per symbol.

The input vector length must be an integer multiple of K. In this configuration, the block accepts a group of K bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol for each group of K bits.

The **Constellation ordering** parameter indicates how the block assigns binary words to points of the signal constellation. Such assignments apply independently to the in-phase and quadrature components of the input:

- If **Constellation ordering** is set to `Binary`, the block uses a natural binary-coded constellation.
- If **Constellation ordering** is set to `Gray` and K is even, the block uses a Gray-coded constellation.
- If **Constellation ordering** is set to `Gray` and K is odd, the block codes the constellation so that pairs of nearest points differ in one or two bits. The constellation is cross-shaped, and the schematic below indicates which pairs of points differ in two bits. The schematic uses M = 128, but suggests the general case.



For details about the Gray coding, see the reference page for the M-PSK Modulator Baseband block and the paper listed in References. Because the in-phase and quadrature components are assigned independently, the Gray and binary orderings coincide when M = 4.

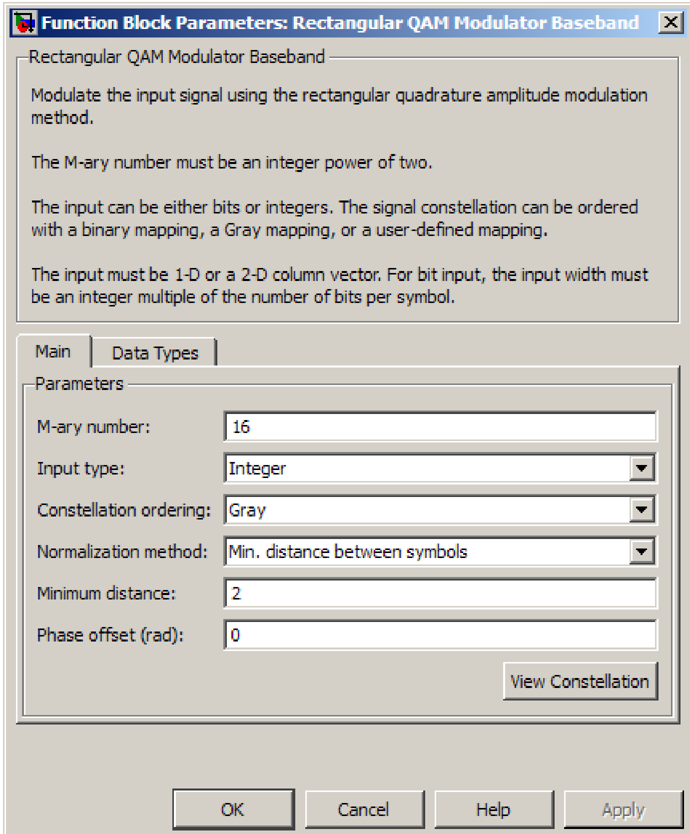
Constellation Size and Scaling

The signal constellation has M points, where M is the **M-ary number** parameter. M must have the form 2^K for some positive integer K. The block scales the signal constellation based on how you set the **Normalization method** parameter. The following table lists the possible scaling conditions.

Value of Normalization Method Parameter	Scaling Condition
`Min.  distance between symbols`	The nearest pair of points in the constellation is separated by the value of the **Minimum distance** parameter
`Average Power`	The average power of the symbols in the constellation is the **Average power** parameter
`Peak Power`	The maximum power of the symbols in the constellation is the **Peak power** parameter

Constellation Visualization

The Rectangular QAM Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. For more information, see the Constellation Visualization section of the *Communications System Toolbox User's Guide*.



Dialog Box

M-ary number

The number of points in the signal constellation. It must have the form 2^K for some positive integer K.

Input type

Indicates whether the input consists of integers or groups of bits.

Constellation ordering

Determines how the block maps each symbol to a group of output bits or integer.

Selecting `User-defined` displays the field **Constellation mapping**, which allows for user-specified mapping.

Constellation mapping

This parameter is a row or column vector of size M and must have unique integer values in the range [0, M-1]. The values must be of data type `double`.

The first element of this vector corresponds to the top-leftmost point of the constellation, with subsequent elements running down column-wise, from left to right. The last element corresponds to the bottom-rightmost point.

This field appears when `User-defined` is selected in the drop-down list **Constellation ordering**.

Normalization method

Determines how the block scales the signal constellation. Choices are `Min. distance between symbols`, `Average Power`, and `Peak Power`.

Minimum distance

The distance between two nearest constellation points. This field appears only when **Normalization method** is set to `Min. distance between symbols`.

Average power, referenced to 1 ohm (watts)

The average power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Average Power`.

Peak power, referenced to 1 ohm (watts)

The maximum power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Peak Power`.

Phase offset (rad)

The rotation of the signal constellation, in radians.

Output data type
The output data type can be set to `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

Setting this parameter to `Fixed-point` or `User-defined` enables fields in which you can further specify details. Setting this parameter to `Inherit via back propagation`, sets the output data type and scaling to match the following block.

Output 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 signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer software. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set output fraction length to
Specify the scaling of the fixed-point output by either of the following 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 **Output 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.

Output 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 output fraction length to** parameter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean when **Input type** is `Bit` • 8-, 16-, 32-bit signed integers • 8-, 16-, 32-bit unsigned integers • $ufix(\log_2 M)$ when **Input type** is `Integer`
Output	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

Rectangular QAM Demodulator Baseband

See Also

General QAM Modulator Baseband

References

[1] Smith, Joel G., “Odd-Bit Quadrature Amplitude-Shift Keying,” *IEEE Transactions on Communications*, Vol. COM-23, March 1975, 385–389.

Purpose

Decode trellis-coded modulation data, modulated using QAM method

Library

TCM, in Digital Baseband sublibrary of Modulation

Description

The Rectangular QAM TCM Decoder block uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using a QAM signal constellation.

The **M-ary number** parameter represents the number of points in the signal constellation, which also equals the number of possible output symbols from the convolutional encoder. (That is, $\log_2$(**M-ary number**) is the number of output bit streams from the convolutional encoder.)

The **Trellis structure** and **M-ary number** parameters in this block should match those in the Rectangular QAM TCM Encoder block, to ensure proper decoding.

Input and Output Signals

This block accepts a column vector input signal containing complex numbers. For information about the data types each block port supports, see "Supported Data Types" on page 2-816.

If the convolutional encoder described by the trellis structure represents a rate *k*/*n* code, then the Rectangular QAM TCM Decoder block's output is a binary column vector with a length of *k* times the vector length of the input signal.

Operation Modes

The block has three possible methods for transitioning between successive frames. The **Operation mode** parameter controls which method the block uses. This parameter also affects the range of possible values for the **Traceback depth** parameter, D.

- In `Continuous` mode, the block initializes all state metrics to zero at the beginning of the simulation, waits until it accumulates D symbols, and then uses a sequence of D symbols to compute each of the traceback paths. D can be any positive integer. At the end of each frame, the block saves its internal state metric for use with the next frame.

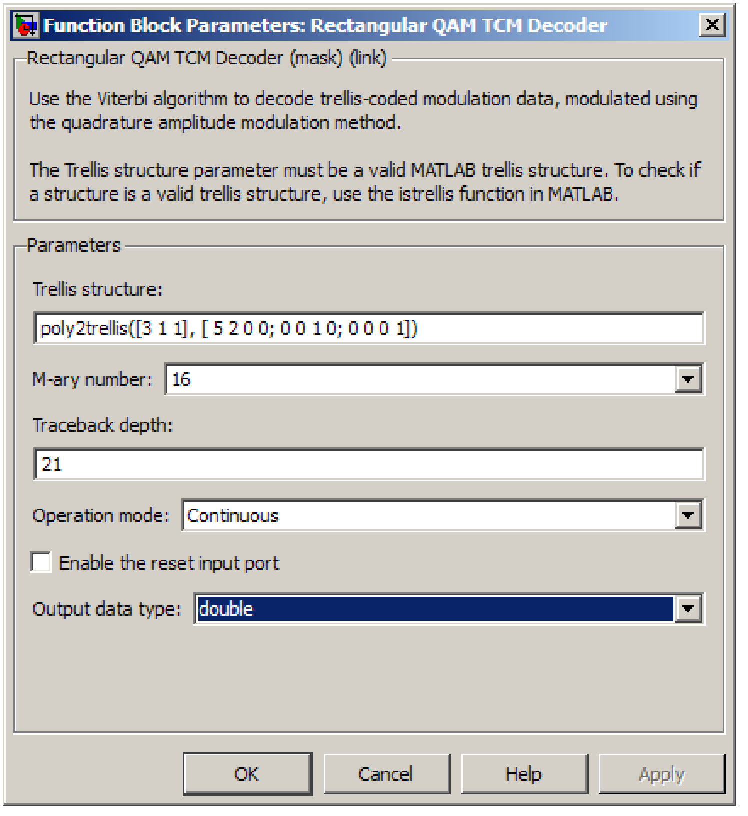
If you select **Enable the reset input**, the block displays another input port, labeled `Rst`. This port receives an integer scalar signal. Whenever the value at the `Rst` port is nonzero, the block resets all state metrics to zero and sets the traceback memory to zero.

- In `Truncated` mode, the block treats each frame independently. The traceback path starts at the state with the lowest metric. D must be less than or equal to the vector length of the input.
- In `Terminated` mode, the block treats each frame independently. The traceback path always starts at the all-zeros state. D must be less than or equal to the vector length of the input. If you know that each frame of data typically ends at the all-zeros state, then this mode is an appropriate choice.

Decoding Delay

If you set **Operation mode** to `Continuous`, then this block introduces a decoding delay equal to **Traceback depth***k bits, for a rate k/n convolutional code. The decoding delay is the number of zeros that precede the first decoded bit in the output.

The block incurs no delay for other values of **Operation mode**.



Dialog Box

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

M-ary number

The number of points in the signal constellation.

Traceback depth

The number of trellis branches (equivalently, the number of symbols) the block uses in the Viterbi algorithm to construct each traceback path.

Operation mode
The operation mode of the Viterbi decoder. Choices are `Continuous`, `Truncated`, and `Terminated`.

Enable the reset input port
When you select this check box, the block has a second input port labeled `Rst`. Providing a nonzero input value to this port causes the block to set its internal memory to the initial state before processing the input data. This option appears only if you set **Operation mode** to `Continuous`.

Output data type
Select the data type for the block output signal as `boolean` or `single`. By default, the block sets this to `double`.

Supported Data Types

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

Pair Block

Rectangular QAM TCM Encoder

See Also

General TCM Decoder, `poly2trellis`

References

[1] Biglieri, E., D. Divsalar, P. J. McLane and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

[2] Proakis, John G., *Digital Communications*, Fourth edition, New York, McGraw-Hill, 2001.

Purpose

Convolutionally encode binary data and modulate using QAM method

Library

TCM, in Digital Baseband sublibrary of Modulation

Description

The Rectangular QAM TCM Encoder block implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal and mapping the result to a QAM signal constellation.

The **M-ary number** parameter is the number of points in the signal constellation, which also equals the number of possible output symbols from the convolutional encoder. (That is, $\log_2$(**M-ary number**) is equal to n for a rate k/n convolutional code.)

Input Signals and Output Signals

If the convolutional encoder described by the trellis structure represents a rate *k*/*n* code, then the Rectangular QAM TCM Encoder block's input must be a binary column vector with a length of *L***k* for some positive integer *L*.

The output from the Rectangular QAM TCM Encoder block is a complex column vector of length *L*.

Specifying the Encoder

To define the convolutional encoder, use the **Trellis structure** parameter. This parameter is a MATLAB structure whose format is described in "Trellis Description of a Convolutional Code" in the Communications System Toolbox documentation. You can use this parameter field in two ways:

- If you want to specify the encoder using its constraint length, generator polynomials, and possibly feedback connection polynomials, then use a `poly2trellis` command within the **Trellis structure** field. For example, to use an encoder with a constraint length of 7, code generator polynomials of 171 and 133 (in octal numbers), and a feedback connection of 171 (in octal), set the **Trellis structure** parameter to

  ```
  poly2trellis(7,[171 133],171)
  ```

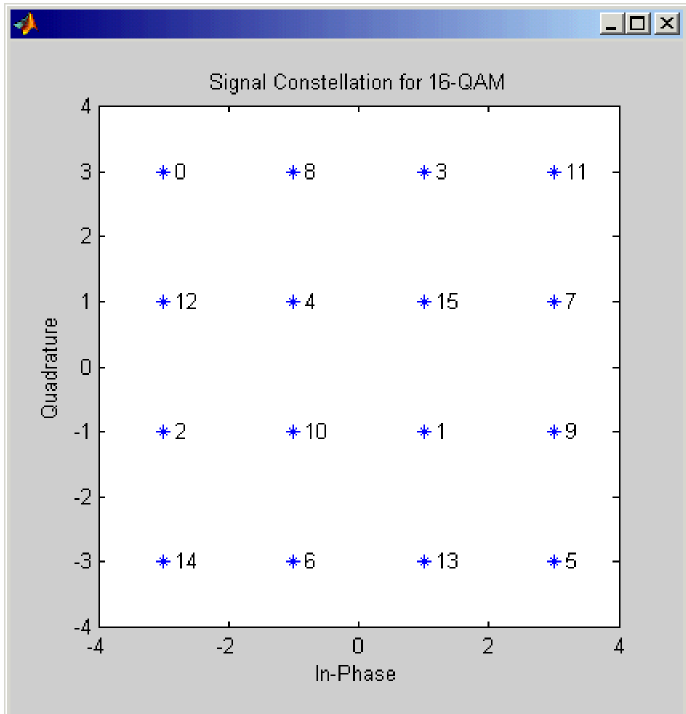
- If you have a variable in the MATLAB workspace that contains the trellis structure, then enter its name as the **Trellis structure** parameter. This way is faster because it causes Simulink to spend less time updating the diagram at the beginning of each simulation, compared to the usage in the previous bulleted item.

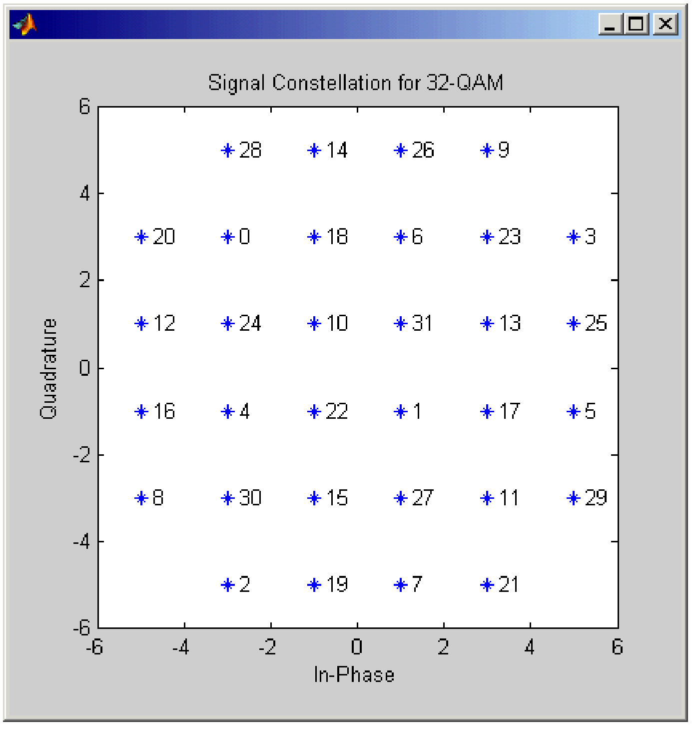
The encoder registers begin in the all-zeros state. You can configure the encoder so that it resets its registers to the all-zeros state during the course of the simulation. To do this, set the `Operation mode` to **Reset on nonzero input via port**. The block then opens a second input port, labeled `Rst`. The signal at the `Rst` port is a scalar signal. When it is nonzero, the encoder resets before processing the data at the first input port.

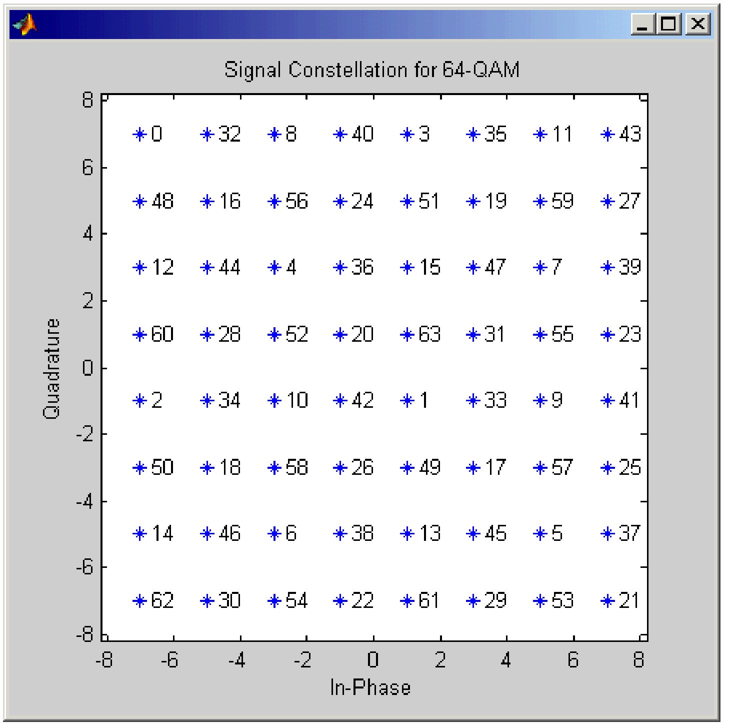
Signal Constellations

The trellis-coded modulation technique partitions the constellation into subsets called cosets, so as to maximize the minimum distance between pairs of points in each coset. This block internally forms a valid partition based on the value you choose for the **M-ary number** parameter.

The figures below show the labeled set-partitioned signal constellations that the block uses when **M-ary number** is `16`, `32`, and `64`. For constellations of other sizes, see Biglieri, E., D. Divsalar, P. J. McLane and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

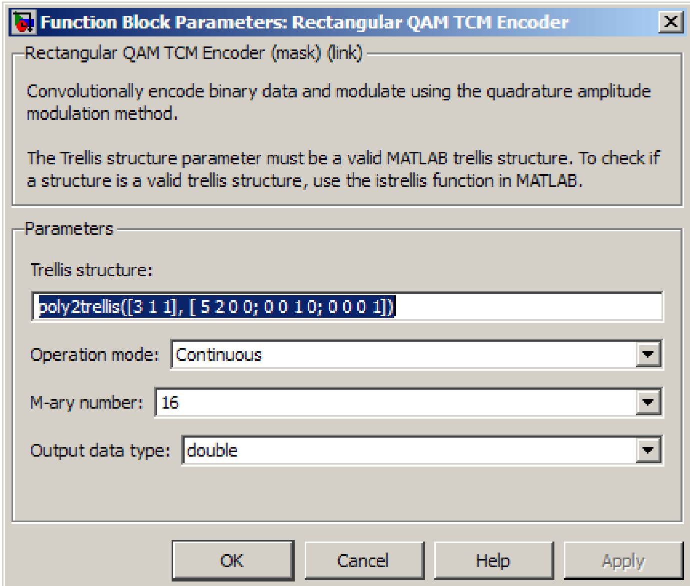

Signal Constellation for 16-QAM
Quadrature
In-Phase
4
3
2
1
0
-1
-2
-3
-4
-4
-2
0
2
4
0
8
3
11
12
4
15
7
2
10
1
9
14
6
13
5


Signal Constellation for 32-QAM
Quadrature
In-Phase
28
14
26
9
20
0
18
6
23
3
12
24
10
31
13
25
16
4
22
1
17
5
8
30
15
27
11
29
2
19
7
21
6
4
2
0
-2
-4
-6
-6
-4
-2
0
2
4
6



Coding Gains

Coding gains of 3 to 6 decibels, relative to the uncoded case can be achieved in the presence of AWGN with multiphase trellis codes. For more information, see Biglieri, E., D. Divsalar, P. J. McLane and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.



Dialog Box

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Operation mode

In `Continuous` mode (default setting), the block retains the encoder states at the end of each frame, for use with the next frame.

In `Truncated (reset every frame)` mode, the block treats each frame independently. I.e., the encoder states are reset to all-zeros state at the start of each frame.

In `Terminate trellis by appending bits` mode, the block treats each frame independently. For each input frame, extra bits are used to set the encoder states to all-zeros state at the end of the frame. The output length is given by

$y = n \cdot (x+s)/k$, where x is the number of input bits, and

$s = \text{constraint length} - 1$ (or, in the case of multiple constraint lengths, s =`sum(ConstraintLength(i)-1)`). The block supports this mode for column vector input signals.

In `Reset on nonzero input via port` mode, the block has an additional input port, labeled `Rst`. When the `Rst` input is nonzero, the encoder resets to the all-zeros state.

M-ary number
The number of points in the signal constellation.

Output data type
The output type of the block can be specified as a `single` or `double`. By default, the block sets this to `double`.

Pair Block

Rectangular QAM TCM Decoder

See Also

General TCM Encoder, `poly2trellis`

References

[1] Biglieri, E., D. Divsalar, P. J. McLane and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001

[3] Ungerboeck, G., “Channel Coding with Multilevel/Phase Signals”, *IEEE Trans. on Information Theory*, Vol IT28, Jan. 1982, pp. 55–67.

Purpose

Resample input at higher rate by repeating values

Library

Signal Operations

Description

The Filter block is a DSP System Toolbox block. For more information, see the Repeat block reference page in the DSP System Toolbox documentation.

Purpose

Generate Rician distributed noise

Library

Noise Generators sublibrary of Comm Sources

Description

The Rician Noise Generator block generates Rician distributed noise. The Rician probability density function is given by

$$f(x) = \begin{cases} \dfrac{x}{\sigma^2} I_0\left(\dfrac{mx}{\sigma^2}\right)\exp\left(-\dfrac{x^2+m^2}{2\sigma^2}\right) & x \geq 0 \\ 0 & x < 0 \end{cases}$$

where:

- σ is the standard deviation of the Gaussian distribution that underlies the Rician distribution noise
- $m^2 = m_I^2 + m_Q^2$, where m_I and m_Q are the mean values of two independent Gaussian components
- I_0 is the modified 0th-order Bessel function of the first kind given by

$$I_0(y) = \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{y\cos t}dt$$

Note that m and σ are *not* the mean value and standard deviation for the Rician noise.

You must specify the **Initial seed** for the random number generator. When it is a constant, the resulting noise is repeatable. The vector length of the Initial seed parameter should equal the number of columns in a frame-based output or the number of elements in a sample-based output. The set of numerical parameters above the **Initial seed** parameter in the dialog box can consist of vectors having the same length as the **Initial seed**, or scalars.

Initial Seed

The scalar **Initial seed** parameter initializes the random number generator that the block uses to generate its Rician-distributed complex random process. For best results, the **Initial seed** should be a prime number greater than 30. Also, if there are other blocks in a model that have an **Initial seed** parameter, you should choose different initial seeds for all such blocks.

You can choose seeds for the Rician Noise Generator block using the Communications System Toolbox randseed function. At the MATLAB prompt, enter

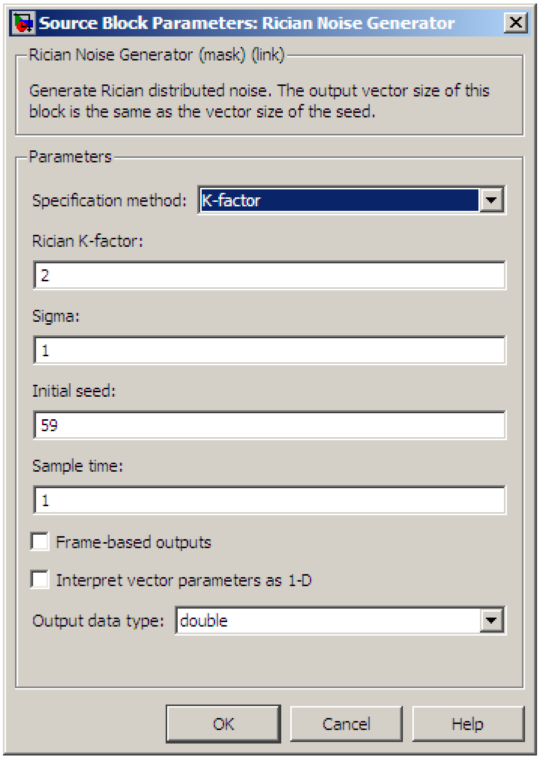
```
randseed
```

This returns a random prime number greater than 30. Entering `randseed` again produces a different prime number. If you supply an integer argument, `randseed` always returns the same prime for that integer. For example, `randseed(5)` always returns the same answer.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples per frame**, and **Interpret vector parameters as 1-D** parameters. See "Sources and Sinks" in *Communications System Toolbox User's Guide* for more details.

The number of elements in the **Initial seed** and **Sigma** parameters becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. Also, the shape (row or column) of the **Initial seed** and **Sigma** parameters becomes the shape of a sample-based two-dimensional output signal.



Dialog Box

Specification method
: Either `K-factor` or `Quadrature components`.

Rician K-factor
: $K = m^2/(2\sigma^2)$, where m is as in the Rician probability density function. This field appears only if **Specification method** is `K-factor`.

In-phase component (mean), Quadrature component (mean)
: The mean values m_I and m_Q, respectively, of the Gaussian components. These fields appear only if **Specification method** is `Quadrature components`.

Sigma
The variable σ in the Rician probability density function.

Initial seed
The initial seed value for the random number generator.

Sample time
The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs
Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame
The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D
If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type
The output can be set to `double` or `single` data types.

See Also

Multipath Rician Fading Channel

References

[1] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

RLS Decision Feedback Equalizer

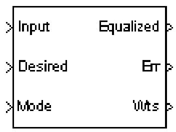
Purpose

Equalize using decision feedback equalizer that updates weights with RLS algorithm

Library

Equalizers

Description



The RLS Decision Feedback Equalizer block uses a decision feedback equalizer and the RLS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the RLS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, the block implements a symbol-spaced equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than `1`, the weights are updated once every N^{th} sample, for a fractionally spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of forward taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input, as described in “Reference Signal and Operation Modes” in *Communications System ToolboxUser’s Guide*.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output, as described in “Adaptive Algorithms” in *Communications System ToolboxUser’s Guide*.

Decision-Directed Mode and Training Mode

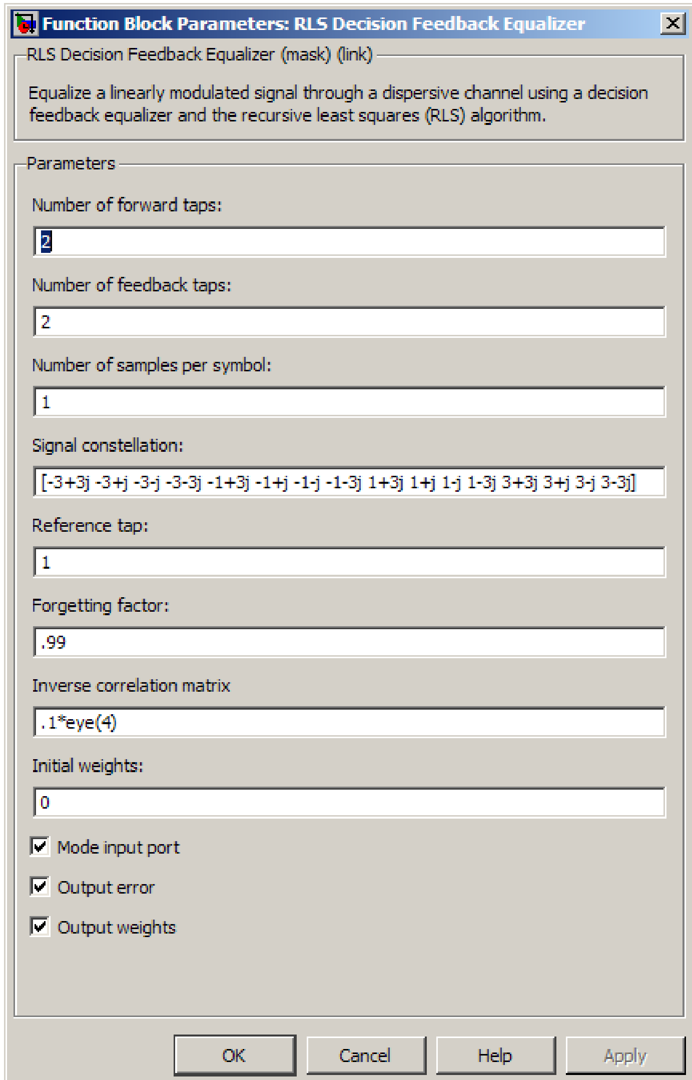
To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap of the forward filter.



Dialog Box

Number of forward taps

The number of taps in the forward filter of the decision feedback equalizer.

Number of feedback taps
The number of taps in the feedback filter of the decision feedback equalizer.

Number of samples per symbol
The number of input samples for each symbol.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulation.

Reference tap
A positive integer less than or equal to the number of forward taps in the equalizer.

Forgetting factor
The forgetting factor of the RLS algorithm, a number between 0 and 1.

Inverse correlation matrix
The initial value for the inverse correlation matrix. The matrix must be N-by-N, where N is the total number of forward and feedback taps.

Initial weights
A vector that concatenates the initial weights for the forward and feedback taps.

Mode input port
When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, and for decision directed, the mode must be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error
When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current forward and feedback weights, concatenated into one vector.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

See Also

RLS Linear Equalizer, LMS Decision Feedback Equalizer, CMA Equalizer

RLS Linear Equalizer

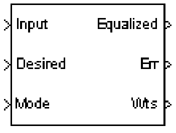
Purpose

Equalize using linear equalizer that updates weights using RLS algorithm

Library

Equalizers

Description



The RLS Linear Equalizer block uses a linear equalizer and the RLS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the RLS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced (i.e. T-spaced) equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than `1`, the block updates the weights once every N^{th} sample, for a fractionally spaced (i.e. T/N-spaced) equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input, as described in “Adaptive Algorithms” in *Communications System Toolbox User’s Guide*.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.

- `Weights` output, as described in "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Decision-Directed Mode and Training Mode

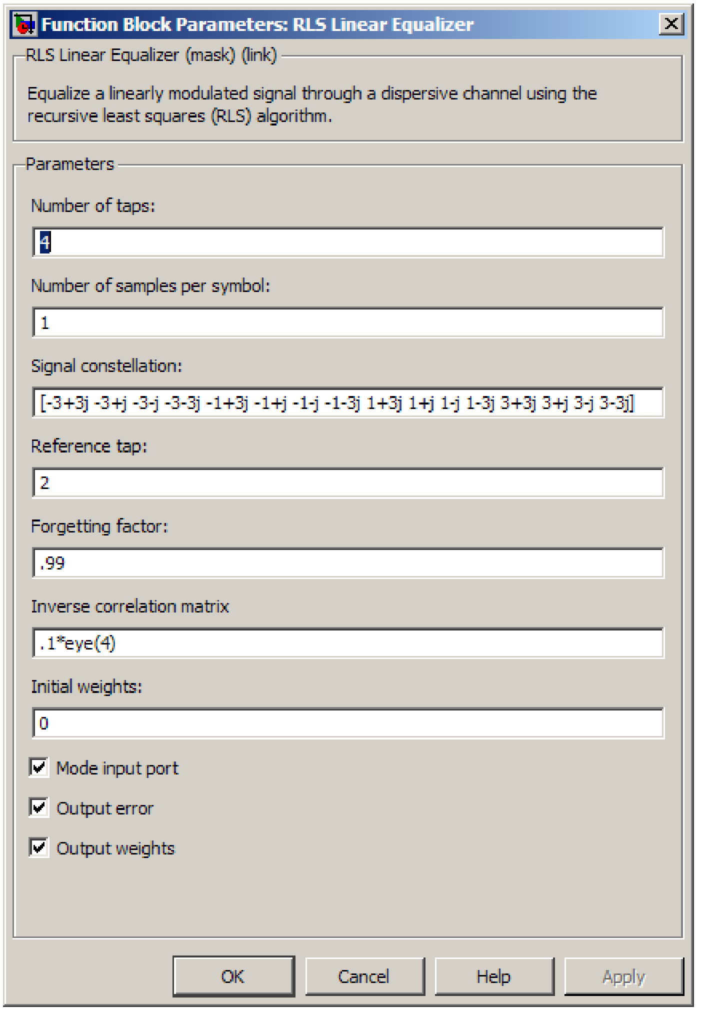
To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap.



Dialog Box

Number of taps

The number of taps in the filter of the linear equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulation.

Reference tap
A positive integer less than or equal to the number of taps in the equalizer.

Forgetting factor
The forgetting factor of the RLS algorithm, a number between 0 and 1.

Inverse correlation matrix
The initial value for the inverse correlation matrix. The matrix must be N-by-N, where N is the number of taps.

Initial weights
A vector that lists the initial weights for the taps.

Mode input port
When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, and for decision directed, the mode must be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error
When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights
When you select this check box, the block outputs the current weights.

Examples

See the Adaptive Equalization example.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

See Also

RLS Decision Feedback Equalizer, LMS Linear Equalizer, CMA Equalizer

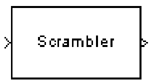
Purpose

Scramble input signal

Library

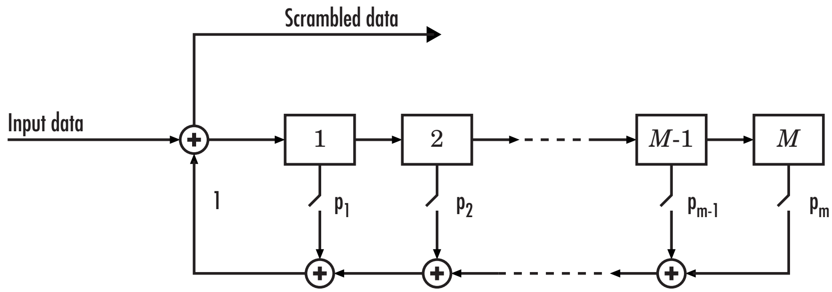
Sequence Operations

Description



The Scrambler block scrambles a scalar or column vector input signal. If you set the **Calculation base** parameter to *N*, then the input values must be integers between 0 and N-1.

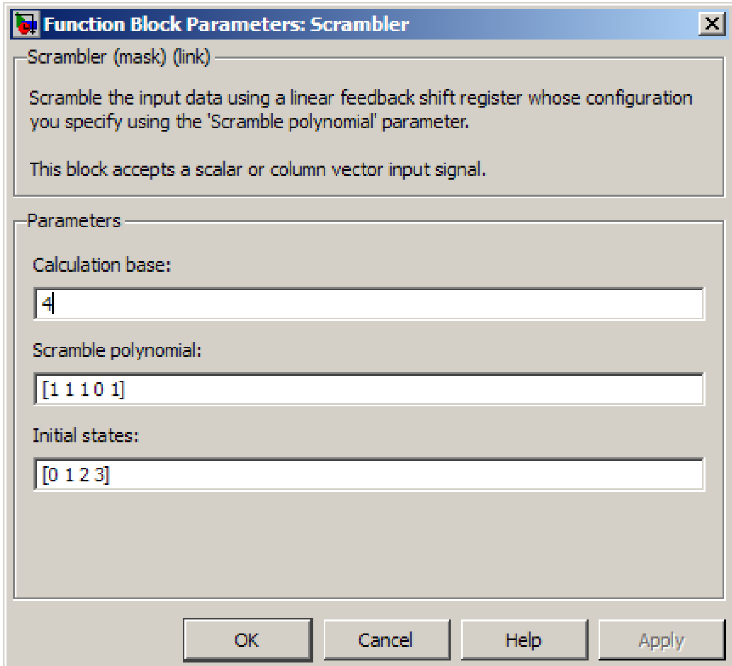
One purpose of scrambling is to reduce the length of strings of 0s or 1s in a transmitted signal, since a long string of 0s or 1s may cause transmission synchronization problems. Below is a schematic of the scrambler. All adders perform addition modulo *N*.



At each time step, the input causes the contents of the registers to shift sequentially. The he **Scramble polynomial** parameter defines if each switch in the scrambler is on or off. Specify the polynomial by listing its coefficients in order of ascending powers of z^{-1}, where $p(z^{-1}) = 1 + p_1z^{-1} + p_2z^{-2}+...$, or by listing the powers of z that appear in the polynomial with a coefficient of 1. For example p = [1 0 0 0 0 0 1 0 1] and p = [0 -6 -8] both represent the polynomial $p(z^{-1}) = 1 + z^{-6} + z^{-8}$.

The **Initial states** parameter lists the states of the scrambler's registers when the simulation starts. The elements of this vector must be integers between 0 and *N*-1. The vector length of this parameter must equal the order of the scramble polynomial. (If the **Scramble polynomial** parameter is a vector that lists the coefficients in order, then the order of the scramble polynomial is one less than the vector length.)

Dialog Box



Calculation base

The calculation base N. The input and output of this block are integers in the range [0, N-1].

Scramble polynomial

A polynomial that defines the connections in the scrambler.

Initial states

The states of the scrambler's registers when the simulation starts.

Pair Block

Descrambler

See Also

PN Sequence Generator

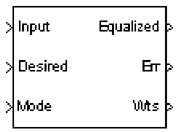
Purpose

Equalize using decision feedback equalizer that updates weights with signed LMS algorithm

Library

Equalizers

Description



The Sign LMS Decision Feedback Equalizer block uses a decision feedback equalizer and an algorithm from the family of signed LMS algorithms to equalize a linearly modulated baseband signal through a dispersive channel.

The supported algorithms, corresponding to the **Update algorithm** parameter, are

- `Sign LMS`
- `Sign Regressor LMS`
- `Sign Sign LMS`

During the simulation, the block uses the particular signed LMS algorithm to update the weights, once per symbol. If the **Number of samples per symbol** parameter is `1`, then the block implements a symbol-spaced equalizer; otherwise, the block implements a fractionally spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of forward taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input, as described in "Reference Signal and Operation Modes" in *Communications System Toolbox User's Guide*.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output, as described in "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Adaptive Algorithms" in *Communications System ToolboxUser's Guide*.

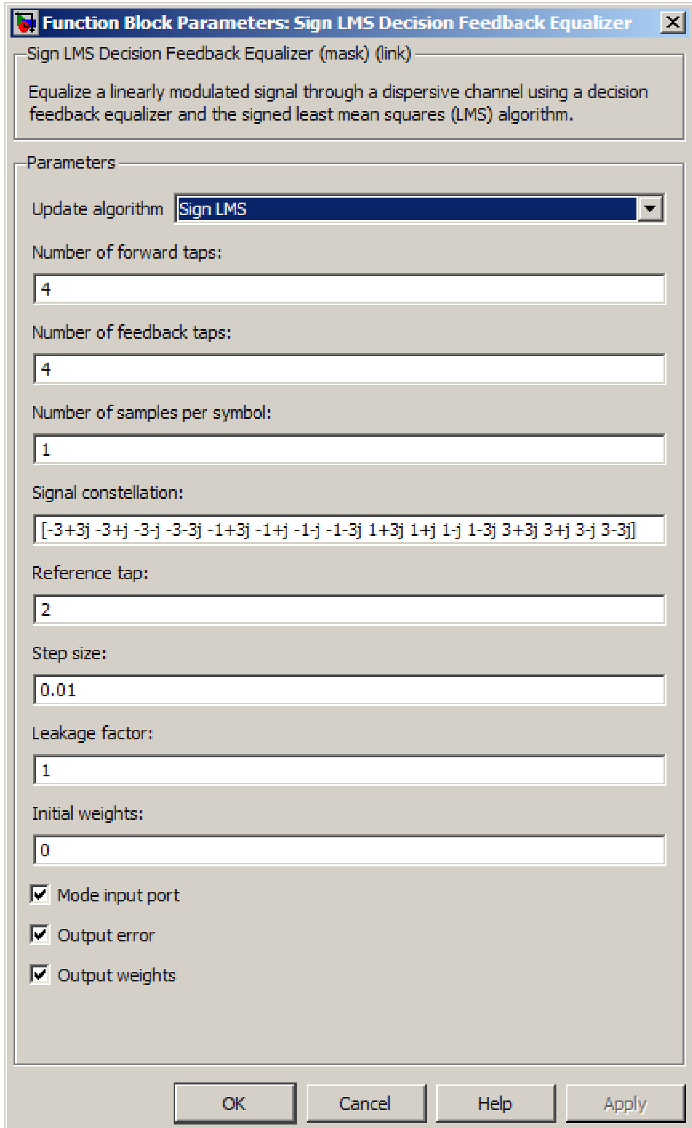
Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap of the forward filter.

Dialog Box



Update algorithm

The specific type of signed LMS algorithm that the block uses to update the equalizer weights.

Number of forward taps
The number of taps in the forward filter of the decision feedback equalizer.

Number of feedback taps
The number of taps in the feedback filter of the decision feedback equalizer.

Number of samples per symbol
The number of input samples for each symbol.

- When you set this parameter to 1, the filter weights are updated once for each symbol, for a symbol spaced (i.e. T-spaced) equalizer.
- When you set this parameter to a value greater than 1, the weights are updated once every N^{th} sample, for a T/N-spaced equalizer.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulation.

Reference tap
A positive integer less than or equal to the number of forward taps in the equalizer.

Step size
The step size of the signed LMS algorithm.

Leakage factor
The leakage factor of the signed LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights
A vector that concatenates the initial weights for the forward and feedback taps.

Mode input port

When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error

When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current forward and feedback weights, concatenated into one vector.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

See Also

Sign LMS Linear Equalizer, LMS Decision Feedback Equalizer

Sign LMS Linear Equalizer

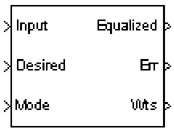
Purpose

Equalize using linear equalizer that updates weights with signed LMS algorithm

Library

Equalizers

Description



The Sign LMS Linear Equalizer block uses a linear equalizer and an algorithm from the family of signed LMS algorithms to equalize a linearly modulated baseband signal through a dispersive channel. The supported algorithms, corresponding to the **Update algorithm** parameter, are

- `Sign LMS`
- `Sign Regressor LMS`
- `Sign Sign LMS`

During the simulation, the block uses the particular signed LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than 1, the weights are updated once every N^{th} sample, for a T/N-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of taps** parameter.

The `Equalized` port outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input, as described in "Adaptive Algorithms" in *Communications System ToolboxUser's Guide*.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output, as described in "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Decision-Directed Mode and Training Mode

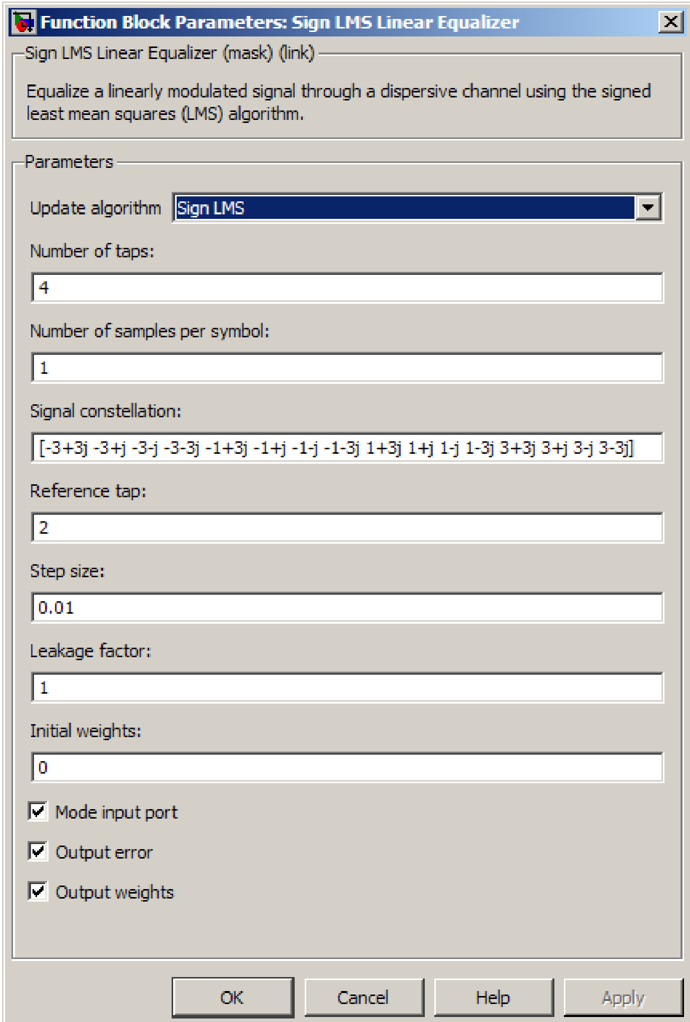
To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap.



Dialog Box

Update algorithm

The specific type of signed LMS algorithm that the block uses to update the equalizer weights.

Number of taps
The number of taps in the filter of the linear equalizer.

Number of samples per symbol
The number of input samples for each symbol.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulation.

Reference tap
A positive integer less than or equal to the number of taps in the equalizer.

Step size
The step size of the signed LMS algorithm.

Leakage factor
The leakage factor of the signed LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights
A vector that lists the initial weights for the taps.

Mode input port
When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error
When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current weights.

Examples

See the Adaptive Equalization example.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

See Also

Sign LMS Decision Feedback Equalizer, LMS Linear Equalizer

Purpose

Recover symbol timing phase using squaring method

Library

Timing Phase Recovery sublibrary of Synchronization

Description

The Squaring Timing Recovery block recovers the symbol timing phase of the input signal using a squaring method. This feedforward, non-data-aided method is similar to the conventional squaring loop. This block is suitable for systems that use linear baseband modulation types such as pulse amplitude modulation (PAM), phase shift keying (PSK) modulation, and quadrature amplitude modulation (QAM).

Typically, the input to this block is the output of a receive filter that is matched to the transmitting pulse shape. This block accepts a column vector input signal of type `double` or `single`. The input represents **Symbols per frame** symbols, using **Samples per symbol** samples for each symbol. Typically, **Symbols per frame** is approximately 100, **Samples per symbol** is at least 4, and the input signal is shaped using a raised cosine filter.

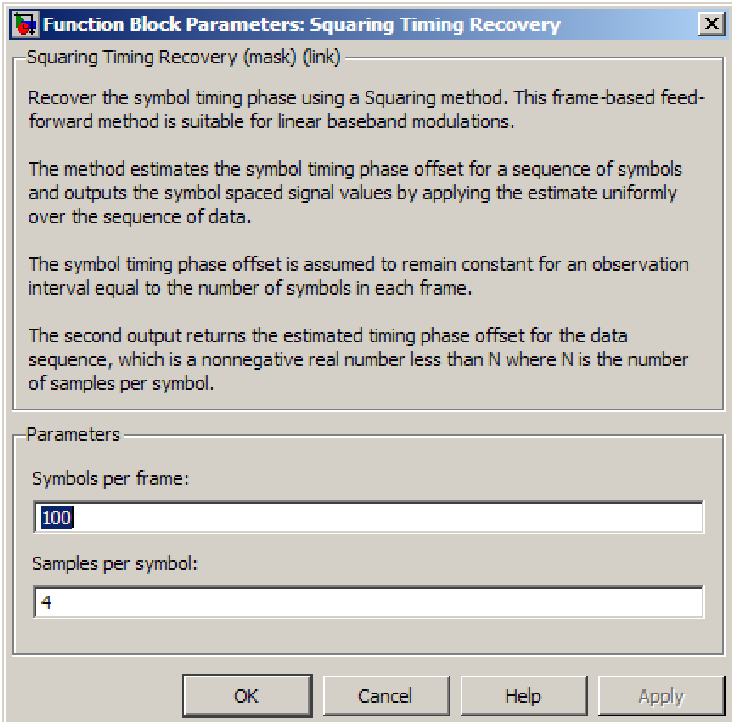
Note The block assumes that the phase offset is constant for all symbols in the entire input frame. If necessary, use the Buffer block to reorganize your data into frames over which the phase offset can be assumed constant. If the assumption of constant phase offset is valid, then a larger frame length yields a more accurate phase offset estimate.

The block estimates the phase offset for the symbols in each input frame and applies the estimate uniformly over the input frame. The block outputs signals containing one sample per symbol. Therefore, the size of each output equals the **Symbols per frame** parameter value. The outputs are as follows:

- The output port labeled `Sym` gives the result of applying the phase estimate uniformly over the input frame. This output is the signal value for each symbol, which can be used for decision purposes.

- The output port labeled `Ph` gives the phase estimate for each symbol in the input frame. All elements in this output are the *same* nonnegative real number less than the **Samples per symbol** parameter value. Noninteger values for the phase estimate correspond to interpolated values that lie between two values of the input signal.

Dialog Box



Symbols per frame

The number of symbols in each frame of the input signal.

Samples per symbol

The number of input samples that represent each symbol. This must be greater than 1.

Algorithm

This block uses a timing estimator that returns

$$-\frac{1}{2\pi}\arg\left(\sum_{m=0}^{\mathrm{LN}-1}\left|x_{m+1}\right|^2\exp(\text{-j}2\pi m/\mathrm{N})\right)$$

as the normalized phase between -1/2 and 1/2, where x is the input vector, L is the **Symbols per frame** parameter and N is the **Samples per symbol** parameter.

For more information about the role that the timing estimator plays in this block's algorithm, see "Feedforward Method for Timing Phase Recovery" in *Communications System Toolbox User's Guide*.

Examples

See "Squaring Timing Phase Recovery Example" in *Communications System Toolbox User's Guide*.

References

[1] Oerder, M. and H. Myer, "Digital Filter and Square Timing Recovery," *IEEE Transactions on Communications*, Vol. COM-36, No. 5, May 1988, pp. 605-612.

[2] Mengali, Umberto and Aldo N. D'Andrea, *Synchronization Techniques for Digital Receivers*, New York, Plenum Press, 1997.

[3] Meyr, Heinrich, Marc Moeneclaey, and Stefan A. Fechtel, *Digital Communication Receivers*, Vol 2, New York, Wiley, 1998.

See Also

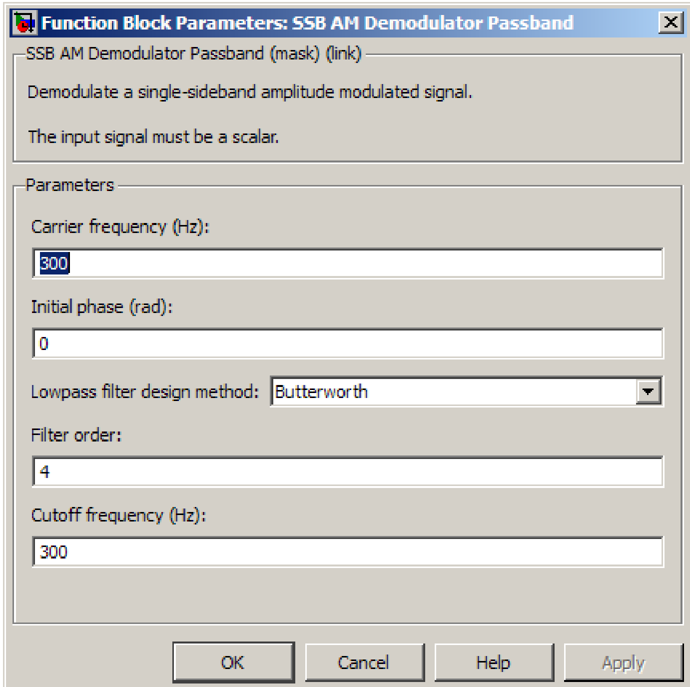
Gardner Timing Recovery, Early-Late Gate Timing Recovery

Purpose Demodulate SSB-AM-modulated data

Library Analog Passband Modulation, in Modulation

Description The SSB AM Demodulator Passband block demodulates a signal that was modulated using single-sideband amplitude modulation. The input is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.



Dialog Box

Carrier frequency (Hz)

The carrier frequency in the corresponding SSB AM Modulator Passband block.

Initial phase (rad)

The phase offset, θ, of the modulated signal.

Lowpass filter design method

The method used to generate the filter. Available methods are Butterworth, Chebyshev type I, Chebyshev type II, and Elliptic.

Filter order

The order of the lowpass digital filter specified in the **Lowpass filter design method** field .

Cutoff frequency
The cutoff frequency of the lowpass digital filter specified in the **Lowpass filter design method** field in Hertz.

Passband ripple
Applies to Chebyshev type I and Elliptic filters only. This is peak-to-peak ripple in the passband in dB.

Stopband ripple
Applies to Chebyshev type II and Elliptic filters only. This is the peak-to-peak ripple in the stopband in dB.

Pair Block

SSB AM Modulator Passband

See Also

DSB AM Demodulator Passband, DSBSC AM Demodulator Passband

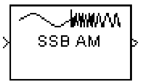
Purpose

Modulate using single-sideband amplitude modulation

Library

Analog Passband Modulation, in Modulation

Description



The SSB AM Modulator Passband block modulates using single-sideband amplitude modulation with a Hilbert transform filter. The output is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

SSB AM Modulator Passband transmits either the lower or upper sideband signal, but not both. To control which sideband it transmits, use the **Sideband to modulate** parameter.

If the input is $u(t)$ as a function of time t, then the output is

$$u(t)\cos(f_c t + \theta) \mp u(t)\sin(f_c t + \theta)$$

where:

- f_c is the **Carrier frequency** parameter.
- θ is the **Initial phase** parameter.
- $\hat{u}(t)$ is the Hilbert transform of the input $u(t)$.
- The minus sign indicates the upper sideband and the plus sign indicates the lower sideband.

Hilbert Tranform Filter

This block uses the Analytic Signal block from the DSP System Toolbox Transforms block library.

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

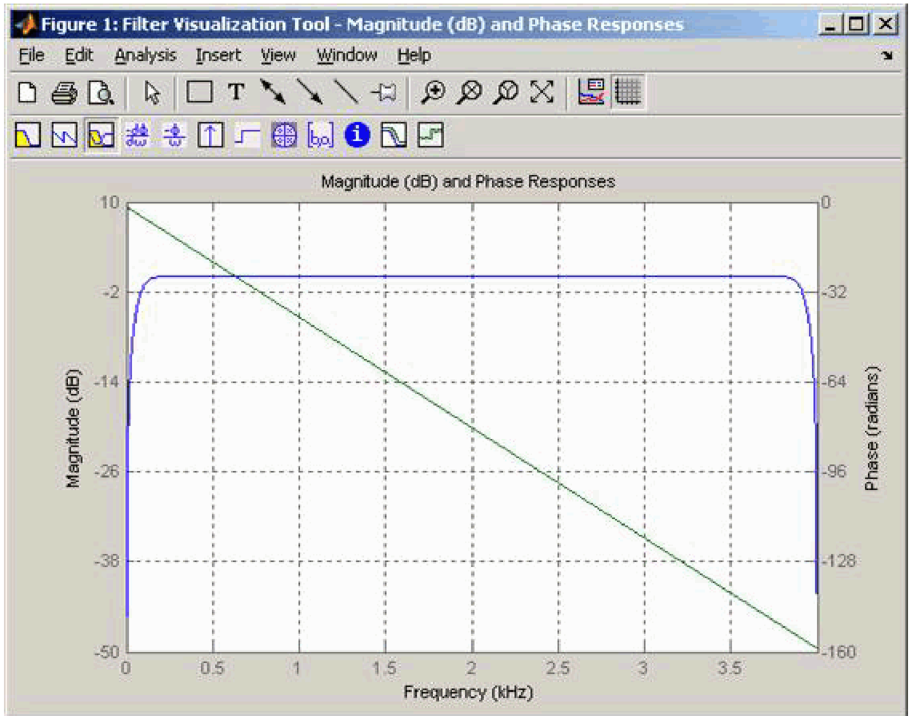
$$y = u + j\mathrm{H}\{u\}$$

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

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

For best results, use a carrier frequency which is estimated to be larger than 10% of your input signal's sample rate. This is due to the implementation of the Hilbert transform by means of a filter.

In the following example, we sample a 10Hz input signal at 8000 samples per second. We then designate a Hilbert Transform filter of order 100. Below is the response of the Hilbert Transform filter as returned by `fvtool`.

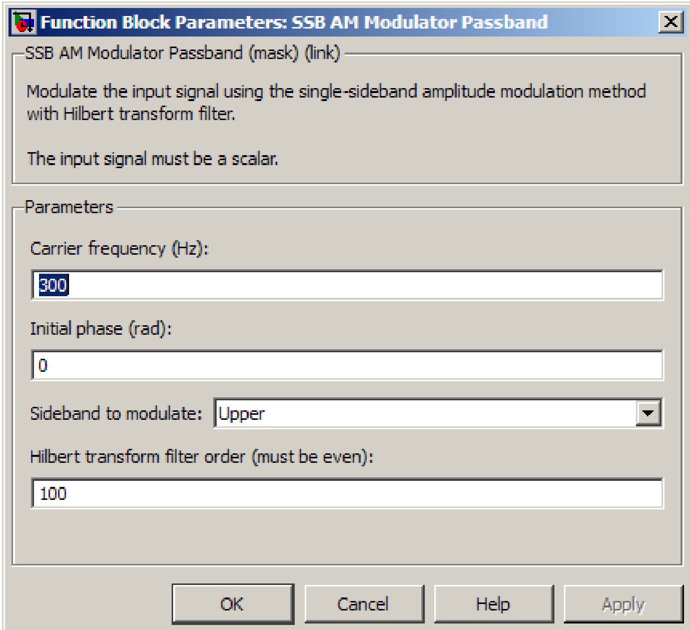


Note the bandwidth of the filter's magnitude response. By choosing a carrier frequency larger than 10% (but less than 90%) of the input signal's sample time (8000 samples per second, in this example) or equivalently, a carrier frequency larger than 400Hz, we ensure that the Hilbert Transform Filter will be operating in the flat section of the filter's magnitude response (shown in blue), and that our modulated signal will have the desired magnitude and form.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Dialog Box



Carrier frequency (Hz)
: The frequency of the carrier.

Initial phase (rad)
: The phase offset, θ, of the modulated signal.

Sideband to modulate
: This parameter specifies whether to transmit the upper or lower sideband.

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

Pair Block

SSB AM Demodulator Passband

See Also

DSB AM Modulator Passband, DSBSC AM Modulator Passband; `hilbiir` (Communications Toolbox)

References

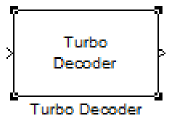
[1] Peebles, Peyton Z, Jr. *Communication System Principles*. Reading, Mass.: Addison-Wesley, 1976.

Turbo Decoder

Purpose Decode input signal using parallel concatenated decoding scheme

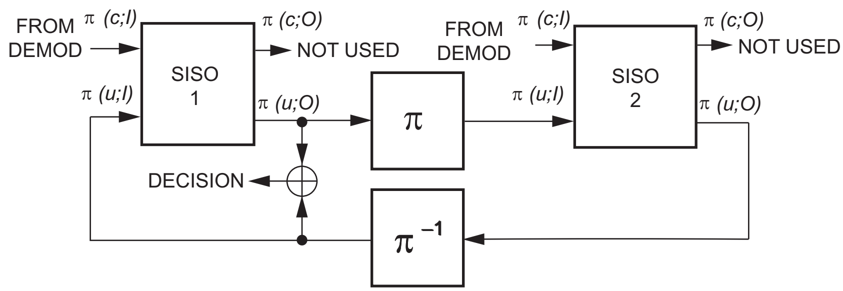
Library Convolutional sublibrary of Error Detection and Correction

Description The Turbo Decoder block decodes the input signal using a parallel concatenated decoding scheme. The iterative decoding scheme uses the *a posteriori* probability (APP) decoder as the constituent decoder, an interleaver, and a deinterleaver.



The two constituent decoders use the same trellis structure and decoding algorithm.

Block Diagram of Iterative Turbo Decoding



The previous block diagram illustrates that the APP decoders (labeled as SISO modules in the previous image) output an updated sequence of log-likelihoods of the encoder input bits, π(u;O). This sequence is based on the received sequence of log-likelihoods of the channel (coded) bits, π(c;I), and code parameters.

The decoder block iteratively updates these likelihoods for a fixed number of decoding iterations and then outputs the decision bits. The interleaver (π) that the decoder uses is identical to the one the encoder uses. The deinterleaver (π^{-1}) performs the inverse permutation with respect to the interleaver. The decoder does not assume knowledge of the tail bits and excludes these bits from the iterations.

Dimensions

This block accepts an M-by-1 column vector input signal and outputs an L-by-1 column vector signal. For a given trellis, L and M are related by:

$$L = \frac{(M - 2 \cdot numTails)}{(2 \cdot n - 1)}$$

and

$$M = L \cdot (2 \cdot n - 1) + 2 \cdot numTails$$

where

M = decoder input length

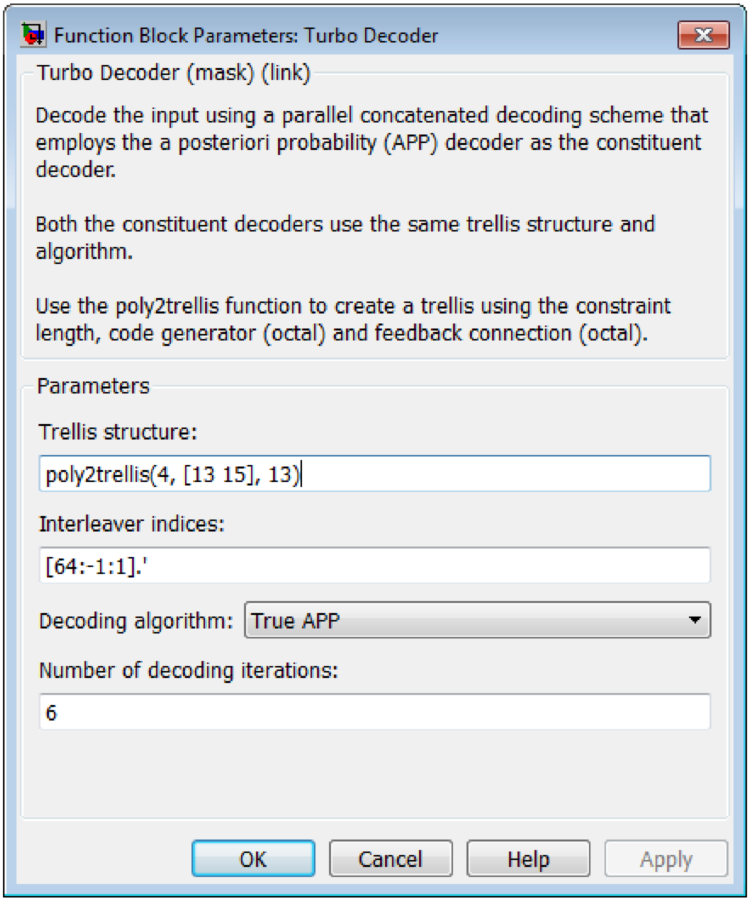
L = decoder output length

n = log2(trellis.NumOutputSymbols), for a rate 1/2 trellis, n = 2

$numTails$ = log2(trellis.numStates) * n

Bit Stream Ordering

The bit ordering subsystem reorganizes the incoming data into the two log likelihood ratio (LLR) streams input to the constituent decoders. This subsystem reconstructs the second systematic stream and reorders the bits so that they match the two constituent encoder outputs at the transmitter. This ordering subsystem is the inverse of the reordering subsystem at the turbo encoder.



Dialog Box

Trellis structure

Trellis structure of constituent convolutional code.

Specify the trellis as a MATLAB structure that contains the trellis description of the constituent convolutional code. Alternatively, use the poly2trellis function to create a custom trellis using the constraint length, code generator (octal), and feedback connection (octal).

The default structure is the result of poly2trellis(4, [13 15], 13).

Interleaver indices

Specify the mapping that the Turbo encoder block uses to permute the input bits as a column vector of integers. The default is `(64:-1:1).'`. This mapping is a vector with the number of elements equal to *L*, the length of the output signal. Each element must be an integer between 1 and *L*, with no repeated values.

Decoding algorithm

Specify the decoding algorithm that the constituent APP decoders use to decode the input signal as `True APP`, `Max*`, `Max`. When you set this parameter to:

- `True APP` – the block implements true *a posteriori* probability decoding
- `Max*` or `Max` – the block uses approximations to increase the speed of the computations.

Number of scaling bits

Specify the number of bits which the constituent APP decoders must use to scale the input data to avoid losing precision during computations. The decoder multiplies the input by 2^**Number of scaling bits** and divides the pre-output by the same factor. The value for this parameter must be a scalar integer between `0` and `8`. This parameter only applies when you set **Decoding algorithm** to `Max*`. The default is `3`.

Number of decoding iterations

Specify the number of decoding iterations the block uses. The default is `6`.

Supported Data Type

Port	Supported Data Types
In	• Double • Single
Out	• Double

Examples

For an example that uses the Turbo Encoder and Turbo Decoder blocks, see the Parallel Concatenated Convolutional Coding: Turbo Codes example.

Pair Block

Turbo Encoder

See Also

APP Decoder

General Block Deinterleaver

General Block Interleaver

References

[1] Berrou, C., A. Glavieux, and P. Thitimajshima. "Near Shannon limit error correcting coding and decoding: turbo codes," *Proceedings of the IEEE International Conference on Communications*, Geneva, Switzerland, May 1993, pp. 1064–1070.

[2] Benedetto, S., G. Montorsi, D. Divsalar, and F. Pollara. " Soft-Input Soft-Output Maximum A Posterior (MAP) Module to Decode Parallel and Serial Concatenated Codes,"*Jet Propulsion Lab TDA Progress Report*, Vol. 42–27, Nov. 1996.

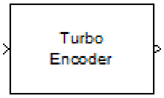
[3] Schlegel, Christian B. and Lance C. Perez. *Trellis and Turbo Coding*, IEEE Press, 2004.

[4] 3GPP TS 36.212 v9.0.0, *3rd Generation partnership project; Technical specification group radio access network; Evolved Universal Terrestrial Radio Acess (E-UTRA); Multiplexing and channel coding (release 9),* 2009-12.

Purpose Encode binary data using parallel concatenated encoding scheme

Library Convolutional sublibrary of Error Detection and Correction

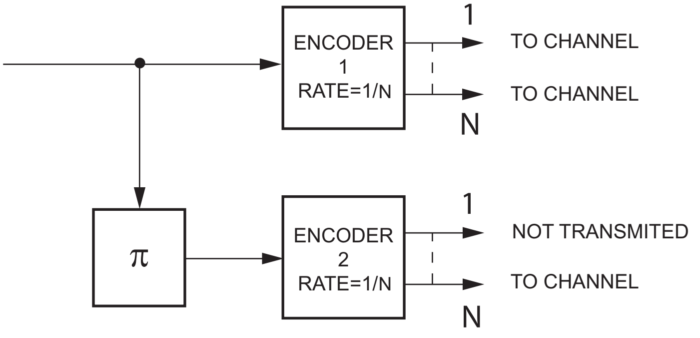
Description



Turbo Encoder

The Turbo Encoder block encodes a binary input signal using a parallel concatenated coding scheme. This coding scheme employs two identical convolutional encoders and one internal interleaver. Each constituent encoder is independently terminated by tail bits.

Block Diagram of Parallel Concatenated Convolutional Code



The previous block diagram illustrates that the output of the Turbo Encoder block consists of the systematic and parity bits streams of the first encoder, and only the parity bit streams of the second encoder.

For a rate one-half constituent encoder, the block interlaces the three streams and multiplexes the tail bits to the end of the encoded data streams.

For more information about tail bits, see the terminate **Operation mode** on the Convolutional Encoder block reference page.

Dimensions

This block accepts an L-by-1 column vector input signal and outputs an M-by-1 column vector signal. For a given trellis, M and L are related by:

$$M = L \cdot (2 \cdot n - 1) + 2 \cdot numTails$$

and

$$L = \frac{(M - 2 \cdot numTails)}{(2 \cdot n - 1)}$$

where

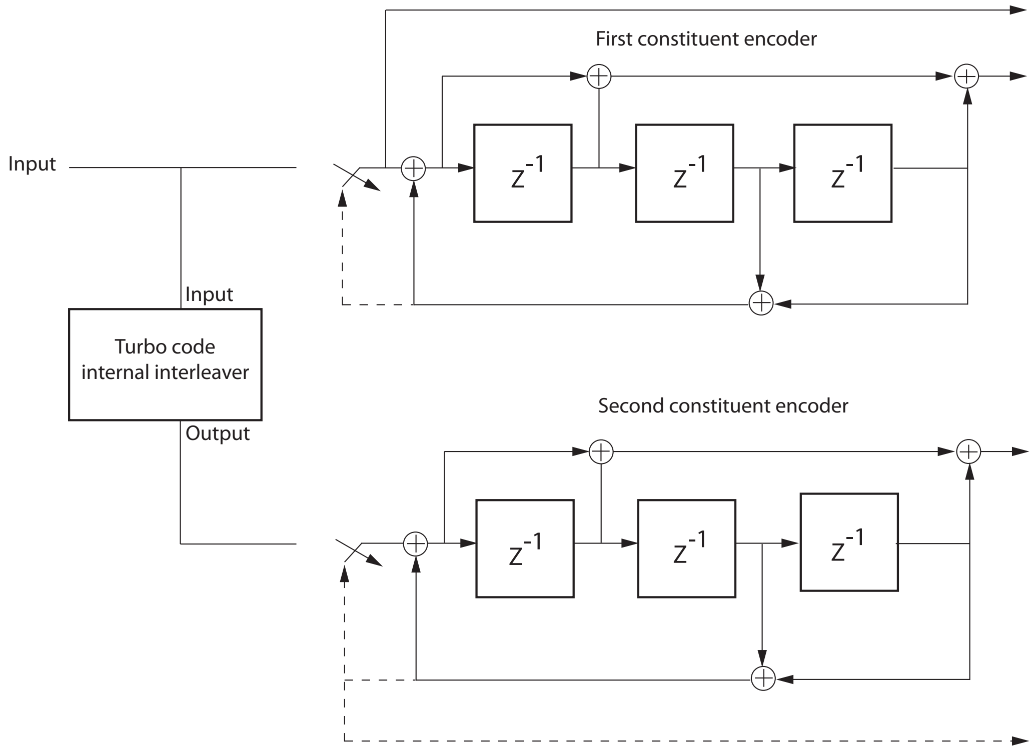
L = encoder input length

M = encoder output length

n = log2(trellis.NumOutputSymbols), for a rate 1/2 trellis, n = 2

$numTails$ = log2(trellis.numStates) * n

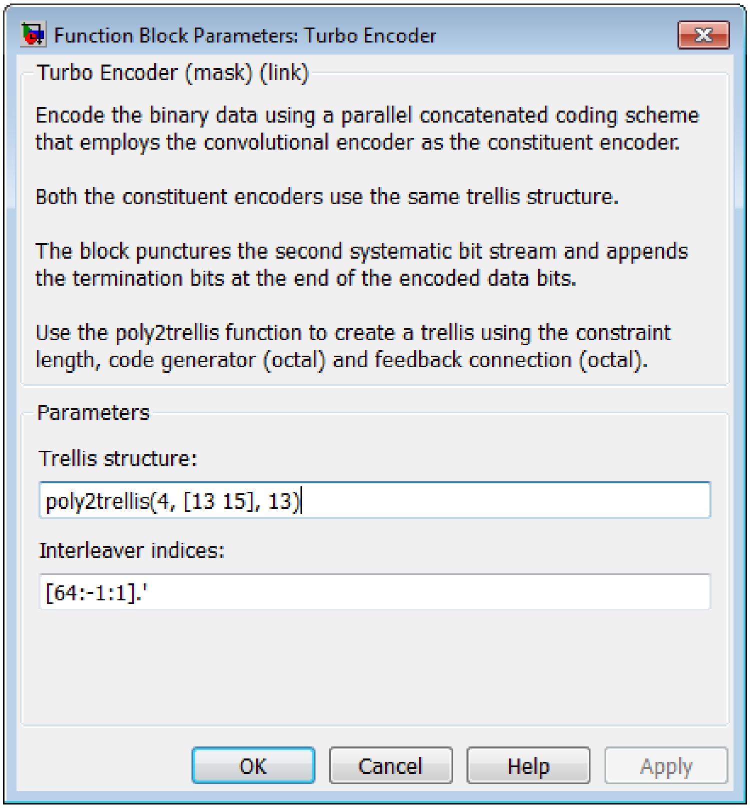
Encoder Schematic for Rate 1/3 Turbo Code Example



The previous schematic shows the encoder configuration for a trellis specified by the default value of the **Trellis structure** parameter, `poly2trellis(4, [13 15], 13)`. For an input vector length of 64 bits, the output of the encoder block is 204 bits. The first 192 bits correspond to the three 64 bit streams (systematic (X_k) and parity (Z_k) bit streams from the first encoder and the parity (Z'_k) bit stream of the second encoder), interlaced as per X_k, Z_k, Z'_k. The last 12 bits correspond to the tail bits from the two encoders, when the switches are in the lower position corresponding to the dashed lines. The first group of six bits are the tail bits from the first constituent encoder and the second group is from the second constituent encoder.

Due to the tail limits, the encoder output code rate is slightly less than 1/3.

Dialog Box



Trellis structure

Trellis structure of constituent convolutional code.

Specify the trellis as a MATLAB structure that contains the trellis description of the constituent convolutional code. Alternatively, use the `poly2trellis` function to create a custom trellis using the constraint length, code generator (octal), and feedback connections (octal).

This block supports only rate 1-by-N trellises where N is an integer.

The default structure is the result of poly2trellis(4, [13 15], 13).

Interleaver indices

Specify the mapping that the block uses to permute the input bits as a column vector of integers. The default is `(64:-1:1).'`. This mapping is a vector with the number of elements equal to the length, L, of the input signal. Each element must be an integer between 1 and L, with no repeated values.

Supported Data Type

Port	Supported Data Types
In	• Double • Single • Fixed-point
Out	• Double • Single • Fixed-point

Examples

For an example that uses the Turbo Encoder and Turbo Decoder blocks, see the Parallel Concatenated Convolutional Coding: Turbo Codes example.

Pair Block

Turbo Decoder

See Also

Convolutional Encoder

General Block Interleaver

References

[1] Berrou, C., A. Glavieux, and P. Thitimajshima. "Near Shannon limit error correcting coding and decoding: turbo codes," *Proceedings of the IEEE International Conference on Communications*, Geneva, Switzerland, May 1993, pp. 1064–1070.

[2] Benedetto, S., G. Montorsi, D. Divsalar, and F. Pollara. " Soft-Input Soft-Output Maximum A Posterior (MAP) Module to Decode Parallel and Serial Concatenated Codes,"*Jet Propulsion Lab TDA Progress Report*, Vol. 42–27, Nov. 1996.

[3] Schlegel, Christian B. and Lance C. Perez. *Trellis and Turbo Coding*, IEEE Press, 2004.

[4] 3GPP TS 36.212 v9.0.0, *3rd Generation partnership project; Technical specification group radio access network; Evolved Universal Terrestrial Radio Acess (E-UTRA); Multiplexing and channel coding (release 9),* 2009-12.

Purpose

Generate uniformly distributed noise between upper and lower bounds

Library

Noise Generators sublibrary of Comm Sources

Description

The Uniform Noise Generator block generates uniformly distributed noise. The output data of this block is uniformly distributed between the specified lower and upper bounds. The upper bound must be greater than or equal to the lower bound.

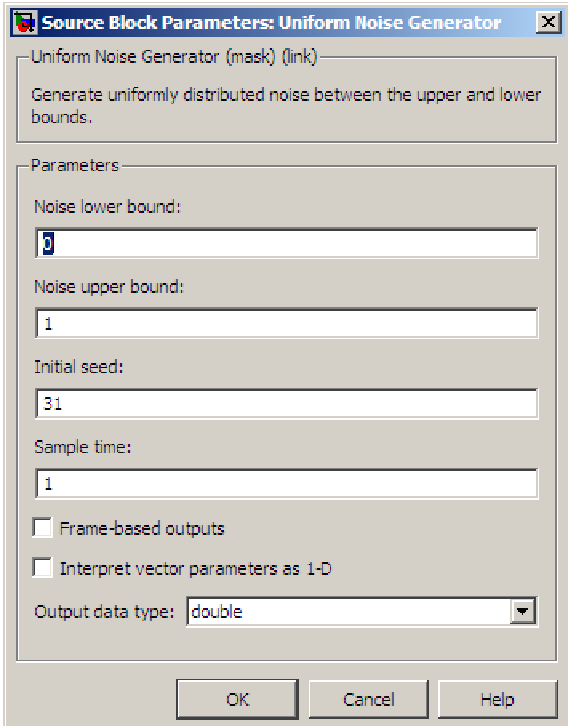
You must specify the **Initial seed** in the simulation. When it is a constant, the resulting noise is repeatable.

If all the elements of the output vector are to be independent and identically distributed (i.i.d.), then you can use a scalar for the **Noise lower bound** and **Noise upper bound** parameters. Alternatively, you can specify the range for each element of the output vector individually, by using vectors for the **Noise lower bound** and **Noise upper bound** parameters. If the bounds are vectors, then their length must equal the length of the **Initial seed** parameter.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples per frame**, and **Interpret vector parameters as 1-D** parameters.

The number of elements in the **Initial seed** parameter becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. Also, the shape (row or column) of the **Initial seed** parameter becomes the shape of a sample-based two-dimensional output signal.



Dialog Box

Noise lower bound, Noise upper bound
: The lower and upper bounds of the interval over which noise is uniformly distributed.

Initial seed
: The initial seed value for the random number generator.

Sample time
: The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs
Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame
The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D
If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type
The output can be set to `double` or `single` data types.

See Also

Random Source (DSP System Toolbox documentation); `rand` (built-in MATLAB function)

Unipolar to Bipolar Converter

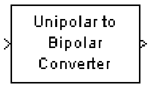
Purpose

Map unipolar signal in range [0, M-1] into bipolar signal

Library

Utility Blocks

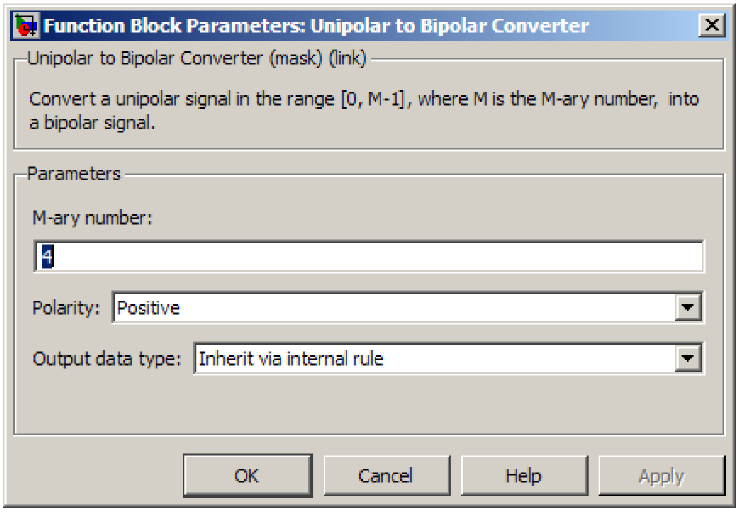
Description



The Unipolar to Bipolar Converter block maps the unipolar input signal to a bipolar output signal. If the input consists of integers between 0 and M-1, where M is the **M-ary number** parameter, then the output consists of integers between -(M-1) and M-1. If M is even, then the output is odd. If M is odd, then the output is even. This block is only designed to work when the input value is within the set {0,1,2...(M-1)}, where M is the **M-ary number** parameter. If the input value is outside of this set of integers the output may not be valid.

The table below shows how the block's mapping depends on the **Polarity** parameter.

Polarity Parameter Value	Output Corresponding to Input Value of k
`Positive`	2k-(M-1)
`Negative`	-2k+(M-1)



Dialog Box

M-ary number

The number of symbols in the bipolar or unipolar alphabet.

Polarity

A value of `Positive` causes the block to maintain the relative ordering of symbols in the alphabets. A value of `Negative` causes the block to reverse the relative ordering of symbols in the alphabets.

Output Data Type

The type of bipolar signal produced at the block's output.

The block supports the following output data types:

- `Inherit via internal rule`
- `Same as input`
- `double`
- `int8`
- `int16`

- `int32`

When the parameter is set to its default setting, `Inherit via internal rule`, the block determines the output data type based on the input data type.

- If the input signal is floating-point (either `single` or `double`), the output data type is the same as the input data type.
- If the input data type is not floating-point:
  - Based on the **M-ary** number parameter, an ideal signed integer output word length required to contain the range [-(M-1)M-1] is computed as follows:

    ideal word length = ceil(log2(M))+1

    Note The +1 is associated with the need for the sign bit.

  - The block sets the output data type to be a signed integer, based on the smallest word length (in bits) that can fit best the computed ideal word length.

Note The selections in the **Hardware Implementation** pane pertaining to word length constraints do not affect how this block determines output data types.

Examples

If the input is [0; 1; 2; 3], the **M-ary number** parameter is `4`, and the **Polarity** parameter is `Positive`, then the output is [-3; -1; 1; 3]. Changing the **Polarity** parameter to `Negative` changes the output to [3; 1; -1; -3].

If the value for the **M-ary number** is 2^7 the block gives an output of int8.

If the value for the **M-ary number** is 2^7+1 the block gives an output of int16.

Pair Block

Bipolar to Unipolar Converter

Variable Step LMS Decision Feedback Equalizer

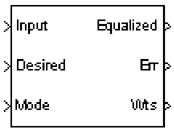
Purpose

Equalize using decision feedback equalizer that updates weights with variable-step-size LMS algorithm

Library

Equalizers

Description



The Variable Step LMS Decision Feedback Equalizer block uses a decision feedback equalizer and the variable-step-size LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the variable-step-size LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than 1, the weights are updated once every N^{th} sample, for a T/N-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of forward taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input, as described in "Reference Signal and Operation Modes" in *Communications System Toolbox User's Guide*.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.

- `Weights` output, as described in "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Decision-Directed Mode and Training Mode

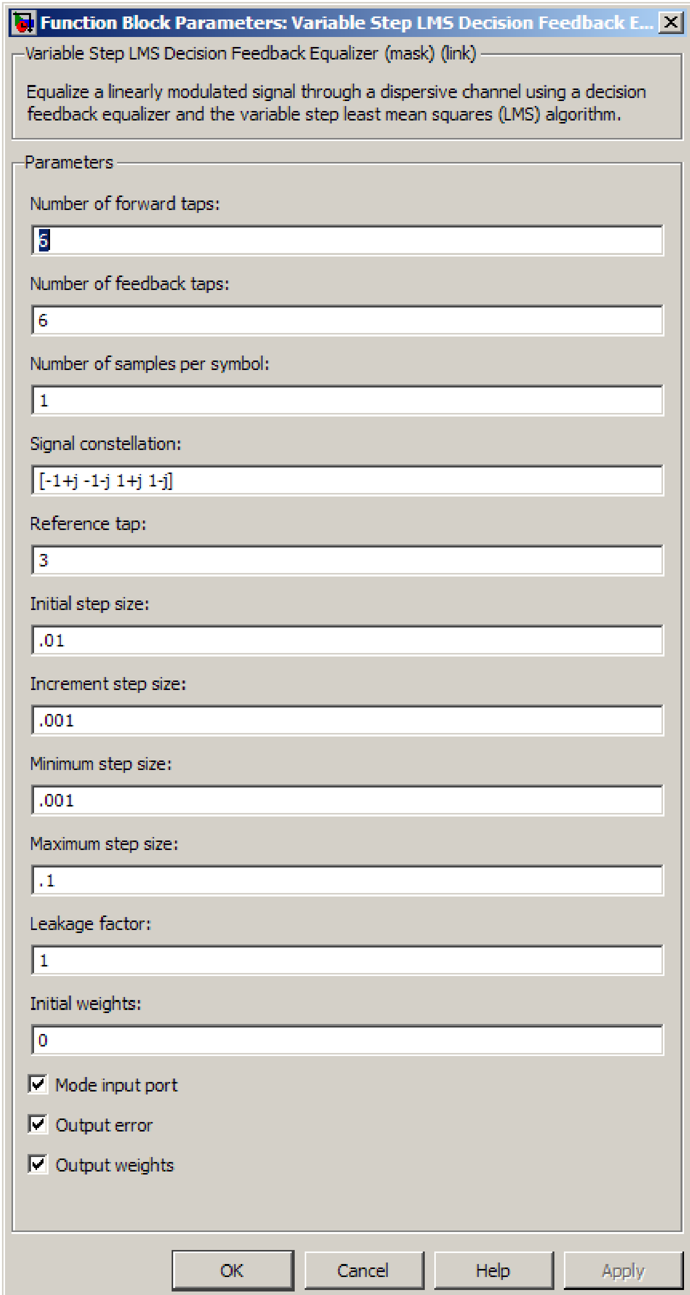
To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap of the forward filter.



Dialog Box

Number of forward taps

The number of taps in the forward filter of the decision feedback equalizer.

Number of feedback taps
The number of taps in the feedback filter of the decision feedback equalizer.

Number of samples per symbol
The number of input samples for each symbol.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulation.

Reference tap
A positive integer less than or equal to the number of forward taps in the equalizer.

Initial step size
The step size that the variable-step-size LMS algorithm uses at the beginning of the simulation.

Increment step size
The increment by which the step size changes from iteration to iteration

Minimum step size
The smallest value that the step size can assume.

Maximum step size
The largest value that the step size can assume.

Leakage factor
The leakage factor of the variable-step-size LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights
A vector that concatenates the initial weights for the forward and feedback taps.

Mode input port
When you select this check box, the block has an input port that enables you to toggle between training and decision-directed

mode. For training, the mode input must be 1, for decision directed, the mode should be 0. The equalizer will train for the length of the Desired signal. If the mode input is not present, the equalizer will train at the beginning of every frame for the length of the Desired signal.

Output error

When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current forward and feedback weights, concatenated into one vector.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

See Also

Variable Step LMS Linear Equalizer, LMS Decision Feedback Equalizer

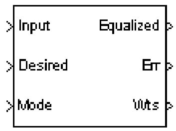
Purpose

Equalize using linear equalizer that updates weights with variable-step-size LMS algorithm

Library

Equalizers

Description



The Variable Step LMS Linear Equalizer block uses a linear equalizer and the variable-step-size LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the variable-step-size LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than 1, the weights are updated once every N^{th} sample, for a T/N-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of taps** parameter.

The `Equalized` port outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input, as described in "Reference Signal and Operation Modes" in *Communications System Toolbox User's Guide*.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output, as described in "Adaptive Algorithms" in *Communications System Toolbox User's Guide*.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see “Adaptive Algorithms” in *Communications System Toolbox User’s Guide*.

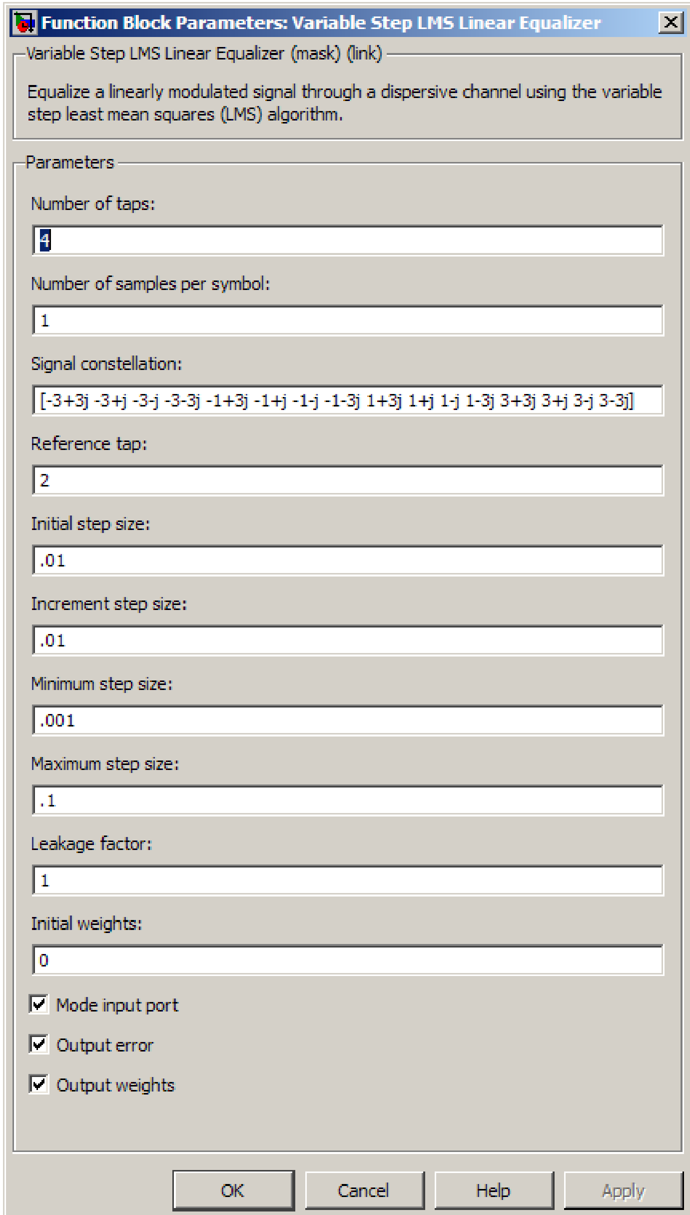
Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter’s modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Since the channel delay is typically unknown, a common practice is to set the reference tap to the center tap.

Dialog Box



Number of taps

The number of taps in the filter of the linear equalizer.

Number of samples per symbol
The number of input samples for each symbol.

Signal constellation
A vector of complex numbers that specifies the constellation for the modulation.

Reference tap
A positive integer less than or equal to the number of taps in the equalizer.

Initial step size
The step size that the variable-step-size LMS algorithm uses at the beginning of the simulation.

Increment step size
The increment by which the step size changes from iteration to iteration

Minimum step size
The smallest value that the step size can assume.

Maximum step size
The largest value that the step size can assume.

Leakage factor
The leakage factor of the LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights
A vector that lists the initial weights for the taps.

Mode input port
When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error

When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current weights.

Examples

See the Adaptive Equalization example.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

See Also

Variable Step LMS Decision Feedback Equalizer, LMS Linear Equalizer

Viterbi Decoder

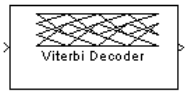
Purpose

Decode convolutionally encoded data using Viterbi algorithm

Library

Convolutional sublibrary of Error Detection and Correction

Description



The Viterbi Decoder block decodes input symbols to produce binary output symbols. This block can process several symbols at a time for faster performance.

This block can output sequences that vary in length during simulation. For more information about sequences that vary in length, or variable-size signals, see “Variable-Size Signal Basics” in the Simulink documentation.

Input and Output Sizes

If the convolutional code uses an alphabet of 2^n possible symbols, this block's input vector length is $L*n$ for some positive integer L. Similarly, if the decoded data uses an alphabet of 2^k possible output symbols, this block's output vector length is $L*k$.

This block accepts a column vector input signal with any positive integer value for L. For variable-sized inputs, the L can vary during simulation. The operation of the block is governed by the operation mode parameter.”

For information about the data types each block port supports, see the “Supported Data Types” on page 2-907 table on this page.

Input Values and Decision Types

The entries of the input vector are either bipolar, binary, or integer data, depending on the **Decision type** parameter.

Decision type Parameter	Possible Entries in Decoder Input	Interpretation of Values	Branch metric calculation
`Unquantized`	Real numbers	Positive real: logical zero Negative real: logical one	Euclidean distance
`Hard Decision`	0, 1	0: logical zero 1: logical one	Hamming distance
`Soft Decision`	Integers between 0 and 2^b-1, where *b* is the **Number of soft decision bits** parameter.	0: most confident decision for logical zero 2^b-1: most confident decision for logical one Other values represent less confident decisions.	Hamming distance

To illustrate the soft decision situation more explicitly, the following table lists interpretations of values for 3-bit soft decisions.

Input Value	Interpretation
0	Most confident zero
1	Second most confident zero
2	Third most confident zero
3	Least confident zero
4	Least confident one

Input Value	Interpretation
5	Third most confident one
6	Second most confident one
7	Most confident one

Operation Modes for Inputs

The Viterbi decoder block has three possible methods for transitioning between successive input frames. The **Operation mode** parameter controls which method the block uses:

- In `Continuous` mode, the block saves its internal state metric at the end of each input, for use with the next frame. Each traceback path is treated independently.
- In `Truncated` mode, the block treats each input independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state. This mode is appropriate when the corresponding Convolutional Encoder block has its **Operation mode** set to `Truncated (reset every frame)`.
- In `Terminated` mode, the block treats each input independently, and the traceback path always starts and ends in the all-zeros state. This mode is appropriate when the uncoded message signal (that is, the input to the corresponding Convolutional Encoder block) has enough zeros at the end of each input to fill all memory registers of the feed-forward encoder. If the encoder has `k` input streams and constraint length vector `constr` (using the polynomial description), "enough" means `k*max(constr-1)`. For feedback encoders, this mode is appropriate if the corresponding Convolutional Encoder block has **Operation mode** set to `Terminate trellis by appending bits`.

Note When this block outputs sequences that vary in length during simulation and you set the **Operation mode** to `Truncated` or `Terminated`, the block's state resets at every input time step.

Use the `Continuous` mode when the input signal contains only one symbol.

Traceback Depth and Decoding Delay

The **Traceback depth** parameter, D, influences the decoding delay. The decoding delay is the number of zero symbols that precede the first decoded symbol in the output.

- If you set the **Operation mode** to `Continuous`, the decoding delay consists of D zero symbols
- If the **Operation mode** parameter is set to `Truncated` or `Terminated`, there is no output delay and the **Traceback depth** parameter must be less than or equal to the number of symbols in each input.

If the code rate is 1/2, a typical **Traceback depth** value is about five times the constraint length of the code.

Reset Port

The reset port is usable only when the **Operation mode** parameter is set to `Continuous`. Selecting **Enable reset input port** gives the block an additional input port, labeled `Rst`. When the `Rst` input is nonzero, the decoder returns to its initial state by configuring its internal memory as follows:

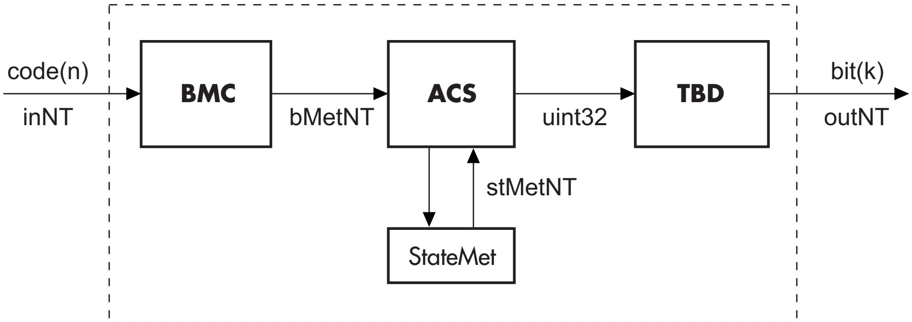
- Sets the all-zeros state metric to zero.
- Sets all other state metrics to the maximum value.
- Sets the traceback memory to zero.

Using a reset port on this block is analogous to setting **Operation mode** in the Convolutional Encoder block to `Reset on nonzero input via port`.

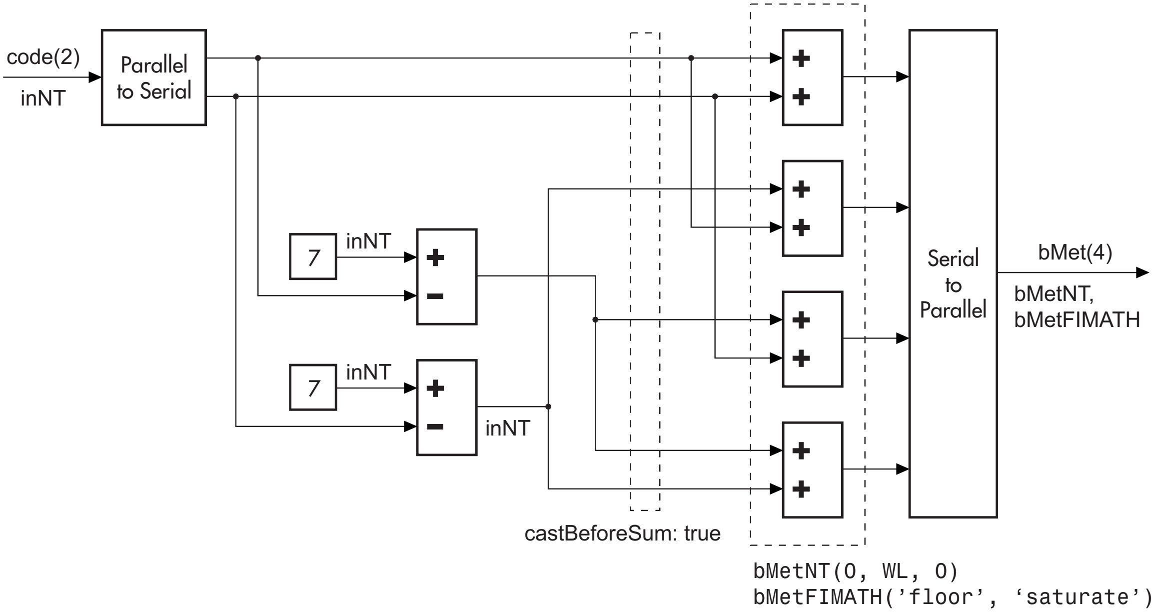
The reset port supports `double` or `boolean` typed signals.

Fixed-Point Signal Flow Diagram

There are three main components to the Viterbi decoding algorithm. They are branch metric computation (BMC), add-compare and select (ACS), and traceback decoding (TBD). The following diagram illustrates the signal flow for a k/n rate code.

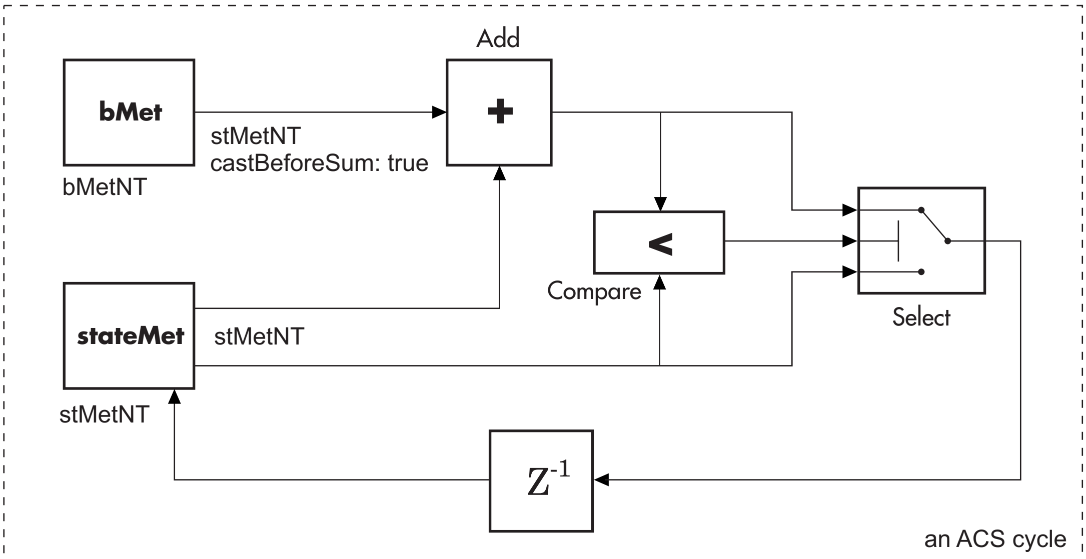


As an example of a BMC diagram, a 1/2 rate, *nsdec* = 3 signal flow would be as follows.



$$WL = nsdec + n - 1$$
$$n = 2 \Rightarrow WL = 4$$

The ACS component is generally illustrated as shown in the following diagram.




```
stMetNT(0, WL2, 0)
stMetFIMATH('floor', 'saturate')
```

Where `WL2` is specified on the mask by the user.

In the flow diagrams above, inNT, bMetNT , stMetNT, and outNT are `numerictype` objects, and bMetFIMATH and stMetFIMATH, are `fimath` objects.

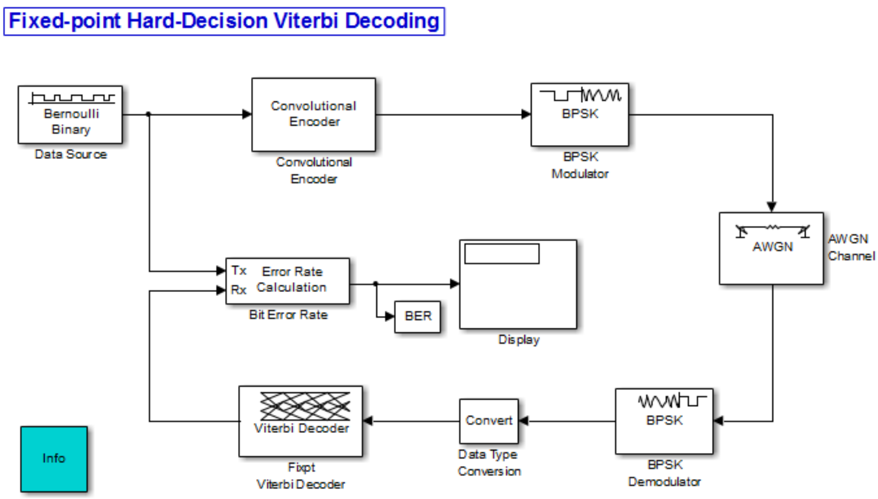
Puncture Pattern Examples

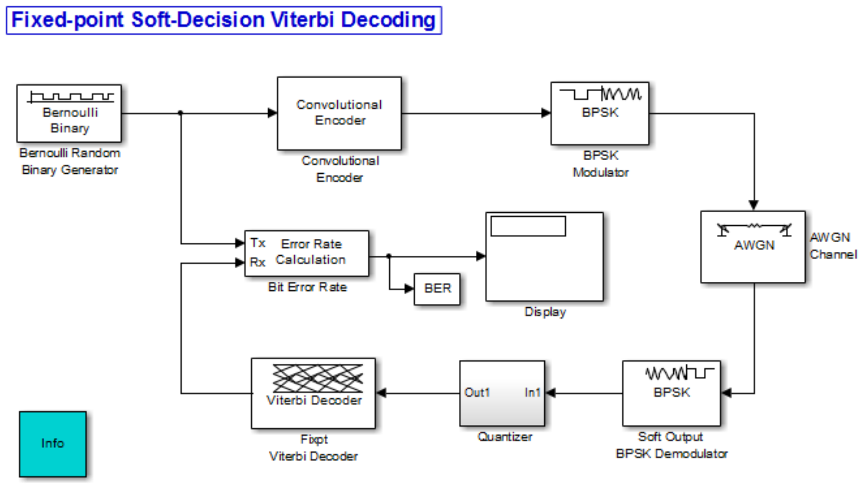
For some commonly used puncture patterns for specific rates and polynomials, see the last three references.

Fixed-Point Viterbi Decoding Examples

The following two example models showcase the fixed-point Viterbi decoder block used for both hard- and soft-decision convolutional decoding.

If you are reading this reference page in the MATLAB Help Browser, click Fixed-point Hard-Decision Viterbi Decoding and Fixed-point Soft-Decision Viterbi Decoding to open the models. These can also be found as `doc_fixpt_vitharddec.mdl` and `doc_fixpt_vitsoftdec.mdl` under `help\toolbox\commm\examples`.





The layout of the soft decision model example is also similar to the existing doc example on Soft-Decision Decoding, which can be found at `help\toolbox\comm\examples\doc_softdecision.mdl`

The purpose of this model is to highlight the fixed-point modeling attributes of the Viterbi decoder, using a familiar layout.

Overview of the Simulations

The two simulations have a similar structure and have most parameters in common. A data source produces a random binary sequence that is convolutionally encoded, BPSK modulated, and passed through an AWGN channel.

The Convolutional encoder is configured as a rate 1/2 encoder. For every 2 bits, the encoder adds another 2 redundant bits. To accommodate this, and add the correct amount of noise, the **Eb/No (dB)** parameter of the AWGN block is in effect halved by subtracting 10*log10(2).

For the hard-decision case, the BPSK demodulator produces hard decisions, at the receiver, which are passed onto the decoder.

For the soft-decision case, the BPSK demodulator produces soft decisions, at the receiver, using the log-likelihood ratio. These soft outputs are 3-bit quantized and passed onto the decoder.

After the decoding, the simulation compares the received decoded symbols with the original transmitted symbols in order to compute the bit error rate. The simulation ends after processing 100 bit errors or 1e6 bits, whichever comes first.

Fixed-Point Modeling

Fixed-point modeling enables bit-true simulations which take into account hardware implementation considerations and the dynamic range of the data/parameters. For example, if the target hardware is a DSP microprocessor, some of the possible word lengths are 8, 16, or 32 bits, whereas if the target hardware is an ASIC or FPGA, there may be more flexibility in the word length selection.

To enable fixed-point Viterbi decoding, the block input must be of type ufix1 (unsigned integer of word length 1) for hard decisions. Based on this input (either a 0 or a 1), the internal branch metrics are calculated using an unsigned integer of word length = (number of output bits), as specified by the trellis structure (which equals 2 for the hard-decision example).

For soft decisions, the block input must be of type ufixN (unsigned integer of word length N), where N is the number of soft-decision bits, to enable fixed-point decoding. The block inputs must be integers in the range 0 to 2^{N-1}. The internal branch metrics are calculated using an unsigned integer of word length = (N + number of output bits - 1), as specified by the trellis structure (which equals 4 for the soft-decision example).

The **State metric word length** is specified by the user and usually must be greater than the branch metric word length already calculated. You can tune this to be the most suitable value (based on hardware and/or data considerations) by reviewing the logged data for the system.

Enable the logging by selecting **Analysis > Fixed-Point Tool**. In the Fixed-Point Setting GUI, set the **Fixed-point instruments mode** to `Minimums, maximums and overflows`, and rerun the simulation. If you see overflows, it implies the data did not fit in the selected container. You could either increase the size of the word length (if your hardware allows it) or try scaling the data prior to processing it. Based on the minimum and maximum values of the data, you are also able to determine whether the selected container is of the appropriate size.

Try running simulations with different values of **State metric word length** to get an idea of its effect on the algorithm. You should be able to narrow down the parameter to a suitable value that has no adverse effect on the BER results.

Comparisons with Double-Precision Data

To run the same model with double precision data, Select **Analysis > Fixed-Point Tool**. In the Fixed-Point Tool GUI, select the **Data type override** to be `Double`. This selection overrides all data type settings in all the blocks to use double precision. For the Viterbi Decoder block, as **Output type** was set to `Boolean`, this parameter should also be set to double.

Upon simulating the model, note that the double-precision and fixed-point BER results are the same. They are the same because the fixed-point parameters for the model have been selected to avoid any loss of precision while still being most efficient.

Comparisons Between Hard and Soft-Decision Decoding

The two models are set up to run from within BERTool to generate a simulation curve that compares the BER performance for hard-decision versus soft-decision decoding.

To generate simulation results for `doc_fixpt_vitharddec.mdl`, do the following:

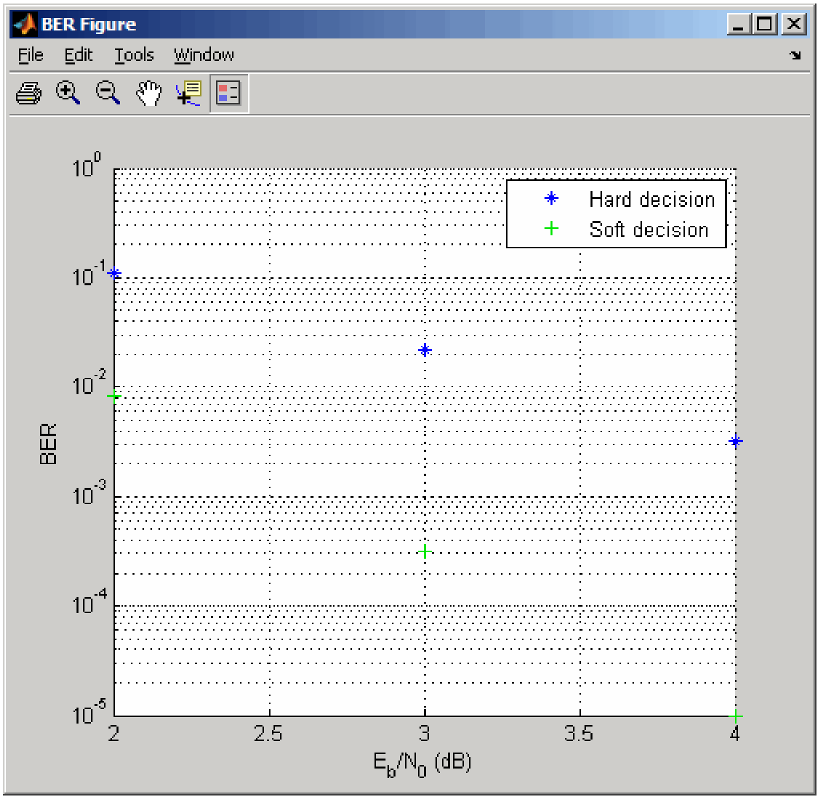
1. Type `bertool` at the MATLAB command prompt.
2. Go to the **Monte Carlo** pane.
3. Set the **Eb/No range** to `2:5`.

4 Set the **Simulation model** to `doc_fixpt_vitharddec.mdl`. Make sure that the model is on path.

5 Set the **BER variable name** to `BER`.

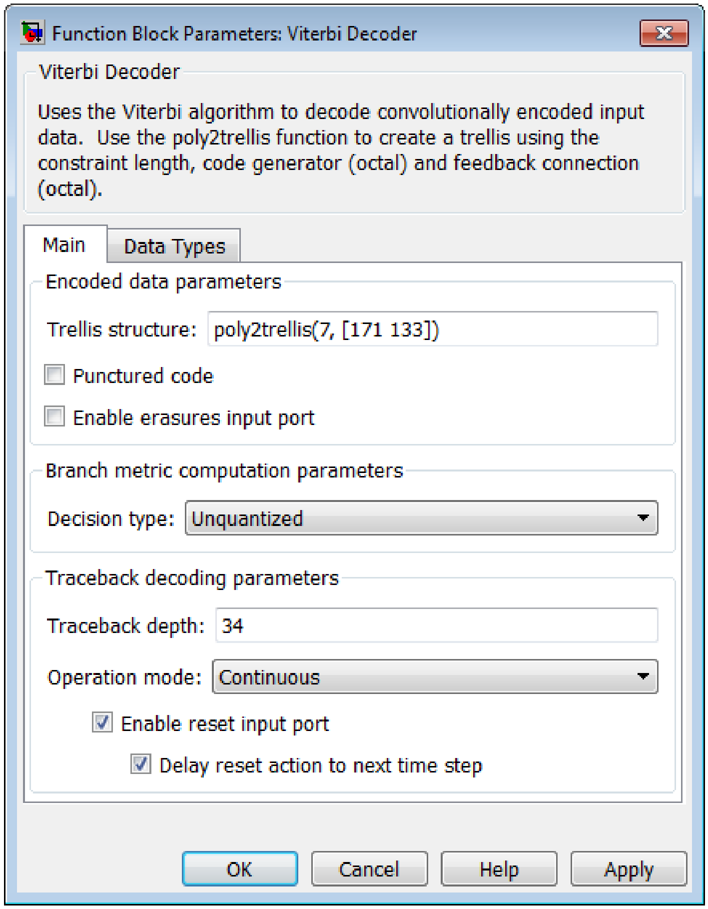
6 Set the **Number of errors** to `100`, and the **Number of bits** to `1e6`.

7 Press **Run** and a plot generates.



To generate simulation results for `doc_fixpt_vitsoftdec.mdl`, just change the **Simulation model** in step 4 and press **Run**.

Notice that, as expected, 3-bit soft-decision decoding is better than hard-decision decoding, roughly to the tune of 1.7 dB, and not 2 dB as commonly cited. The difference in the expected results could be attributed to the imperfect quantization of the soft outputs from the demodulator.



Dialog Box

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder. Use the same value here and in the corresponding Convolutional Encoder block.

Punctured code

Select this check box to specify a punctured input code. The field, **Punctured code**, appears.

Puncture vector

Constant puncture pattern vector used at the transmitter (encoder). The puncture vector is a pattern of `1`s and `0`s. The `0`s indicate the punctured bits. When you select **Punctured code**, the **Punctured vector** field appears.

Enable erasures input port

When you check this box, the decoder opens an input port labeled `Era`. Through this port, you can specify an erasure vector pattern of `1`s and `0`s, where the `1`s indicate the erased bits.

For these erasures in the incoming data stream, the decoder does not update the branch metric. The widths and the sample times of the erasure and the input data ports must be the same. The erasure input port can be of data type `double` or `Boolean`.

Decision type

Specifies the use of `Unquantized`, `Hard Decision`, or `Soft Decision` for the branch metric calculation.

- `Unquantized` decision uses the Euclidean distance to calculate the branch metrics.
- `Soft Decision` and `Hard Decision` use the Hamming distance to calculate the branch metrics, where **Number of soft decision bits** equals `1`.

Number of soft decision bits

The number of soft decision bits to represent each input. This field is active only when **Decision type** is set to `Soft Decision`.

Error if quantized input values are out of range

Select this check box to throw an error when quantized input values are out of range. This check box is active only when **Decision type** is set to `Soft Decision` or `Hard Decision`.

Traceback depth

The number of trellis branches to construct each traceback path.

Operation mode

Method for transitioning between successive input frames: `Continuous`, `Terminated`, and `Truncated`.

Note When this block outputs sequences that vary in length during simulation and you set the **Operation mode** to `Truncated` or `Terminated`, the block's state resets at every input time step.

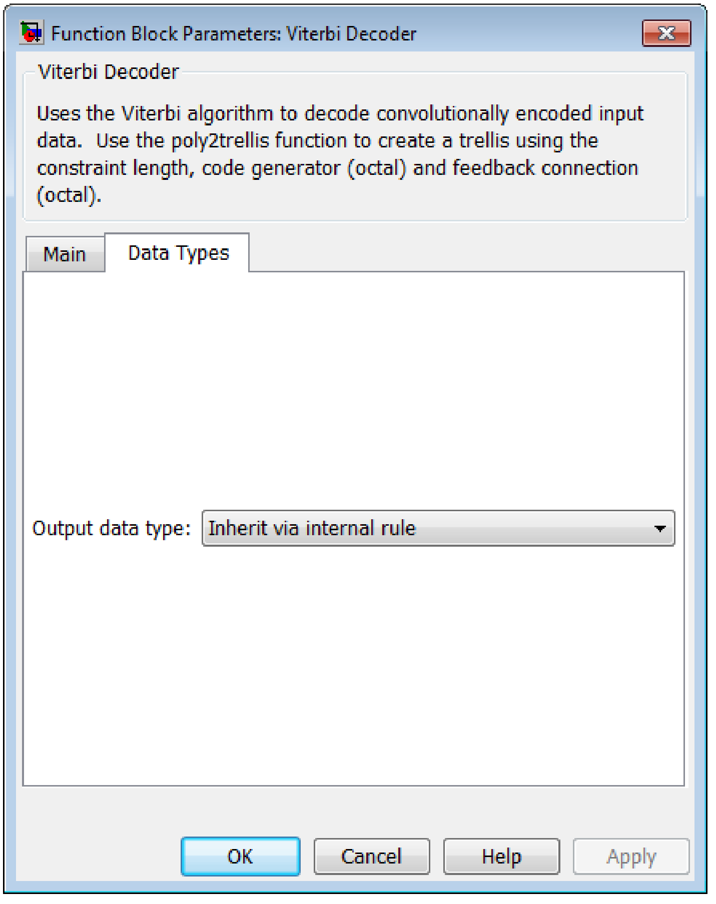
Enable reset input port

When you check this box, the decoder opens an input port labeled `Rst`. Providing a nonzero input value to this port causes the block to set its internal memory to the initial state before processing the input data.

Delay reset action to next time step

When you select this option, the Viterbi Decoder block resets after decoding the encoded data. This option is available only when you set **Operation mode** to `Continuous` and select **Enable reset input port**. You must enable this option for HDL support.

Output data type



The output signal's data type can be `double`, `single`, `boolean`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or set to `'Inherit via internal rule'` or `'Smallest unsigned integer'`.

When set to `'Smallest unsigned integer'`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type is `ufix(1)`. For all other selections, it is an unsigned integer with the smallest specified wordlength corresponding to the char value (e.g., `uint8`).

When set to `'Inherit via internal rule'` (the default setting), the block selects double-typed outputs for double inputs, single-typed outputs for single inputs, and behaves similarly to the `'Smallest unsigned integer'` option for all other typed inputs.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean for `Hard decision` mode • 8-, 16-, and 32-bit signed integers (for `Hard decision` and `Soft decision` modes) • 8-, 16-, and 32-bit unsigned integers (for `Hard decision` and `Soft decision` modes) • ufix(n), where n represents the **Number of soft decision bits**
Output	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) for `ASIC/FPGA` mode

See Also

Convolutional Encoder, APP Decoder

References

[1] Clark, G. C. Jr. and J. Bibb Cain., *Error-Correction Coding for Digital Communications*, New York, Plenum Press, 1981.

[2] Gitlin, R. D., J. F. Hayes, and S. B. Weinstein, *Data Communications Principles*, New York, Plenum, 1992.

[3] Heller, J. A. and I. M. Jacobs, "Viterbi Decoding for Satellite and Space Communication," *IEEE Transactions on Communication Technology*, Vol. COM-19, October 1971, pp 835–848.

[4] Yasuda, Y., et. al., "High-rate punctured convolutional codes for soft decision Viterbi decoding," *IEEE Transactions on Communications*, Vol. COM-32, No. 3, pp 315–319, March 1984.

[5] Haccoun, D., and Begin, G., "High-rate punctured convolutional codes for Viterbi and sequential decoding," *IEEE Transactions on Communications*, Vol. 37, No. 11, pp 1113–1125, Nov. 1989.

[6] Begin, G., et.al., "Further results on high-rate punctured convolutional codes for Viterbi and sequential decoding," *IEEE Transactions on Communications*, Vol. 38, No. 11, pp 1922–1928, Nov. 1990.

Walsh Code Generator

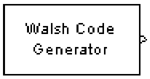
Purpose

Generate Walsh code from orthogonal set of codes

Library

Sequence Generators sublibrary of Comm Sources

Description



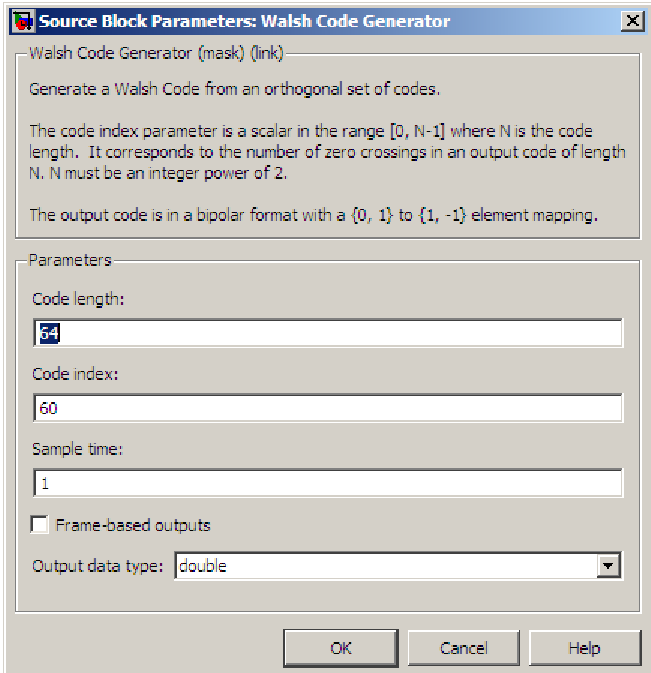
Walsh codes are defined as a set of N codes, denoted W_j, for $j = 0, 1, \ldots$, N - 1, which have the following properties:

- W_j takes on the values +1 and -1.
- $W_j[0] = 1$ for all j.
- W_j has exactly j zero crossings, for $j = 0, 1, \ldots , N - 1$.
- $W_j W_k^T = \begin{cases} 0 & j \neq k \\ N & j = k \end{cases}$
- Each code Wj is either even or odd with respect to its midpoint.

Walsh codes are defined using a Hadamard matrix of order N. The Walsh Code Generator block outputs a row of the Hadamard matrix specified by the **Walsh code index**, which must be an integer in the range [0, ..., N - 1]. If you set **Walsh code index** equal to an integer j, the output code has exactly j zero crossings, for $j = 0, 1, \ldots , N - 1$.

Note, however, that the indexing in the Walsh Code Generator block is different than the indexing in the Hadamard Code Generator block. If you set the **Walsh code index** in the Walsh Code Generator block and the **Code index parameter** in the Hadamard Code Generator block, the two blocks output different codes.

Dialog Box



Code length

Integer scalar that is a power of 2 specifying the length of the output code.

Code index

Integer scalar in the range [0, 1, ... , N - 1], where N is the **Code length**, specifying the number of zero crossings in the output code.

Sample time

A positive real scalar specifying the sample time of the output signal.

Frame-based outputs

When checked, the block outputs a frame-based signal. When cleared, the block outputs a [1] unoriented scalar.

Samples per frame
The number of samples in a frame-based output signal. This field is active only if you select **Frame-based outputs**. If **Samples per frame** is greater than the **Code length**, the code is cyclically repeated.

Output data type
The output type of the block can be specified as an `int8` or `double`. By default, the block sets this to `double`.

See Also

Hadamard Code Generator, OVSF Code Generator

Windowed Integrator

Purpose

Integrate over time window of fixed length

Library

Comm Filters

Description

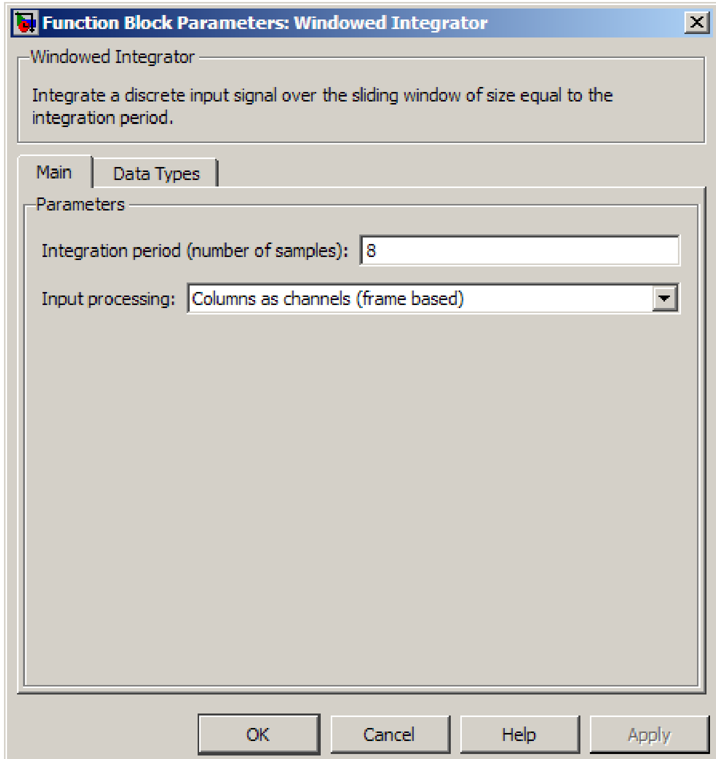
The Windowed Integrator block creates cumulative sums of the input signal values over a sliding time window of fixed length. If the **Integration period** parameter is N and the input samples are denoted by x(1), x(2), x(3),..., then the nth output sample is the sum of the x(k) values for k between n-N+1 and n. In cases where n-N+1 is less than 1, the block uses an initial condition of 0 to represent those samples.

Input and Output Signals

This block accepts scalar, column vector, and *M*-by-*N* matrix input signals. 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 *N* instances of the same filter, each with its own independent state buffer. Each of the *N* filters process *M* 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. For information about the data types each block port supports, see the "Supported Data Type" on page 2-918 table on this page.



Dialog Box

Integration period

The length of the interval of integration, measured in samples.

Input processing

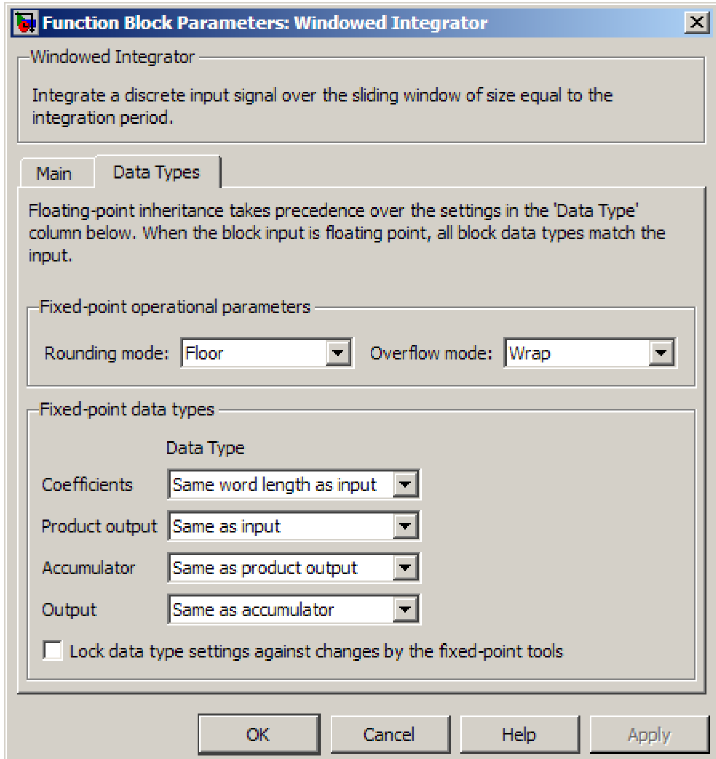
Specify how the block processes the input signal. You can set this parameter to one of the following options:

- `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 *Communications System Toolbox Release Notes* for more information.

This parameter is available only when you set the **Rate options** parameter to `Allow multirate processing`.



Rounding mode

Select the rounding mode for fixed-point operations. The block uses the **Rounding mode** when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. The filter coefficients do not obey this parameter; they always round to `Nearest`. For more information, see “Rounding Modes” in the DSP System Toolbox documentation or “Rounding Mode: Simplest” in the Fixed-Point Designer documentation.

Overflow mode

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

Coefficients

The block implementation uses a Direct-Form FIR filter with all tap weights set to one. The **Coefficients** parameter controls which data type represents the taps (i.e. ones) when the input data is a fixed-point signal.

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). See "Filter Structure Diagrams" in *DSP System Toolbox Reference Guide* 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 are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits. If applicable, you are able to enter separate fraction lengths for the numerator and denominator coefficients.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. If applicable, you are able to enter separate slopes for the

numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.

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

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See “Filter Structure Diagrams” and “Multiplication Data Types” in *DSP System Toolbox Reference Guide* 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 are able to enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See “Filter Structure Diagrams” 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 are able to enter the word length and the fraction length of the accumulator, in bits.

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

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

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling tool in the Fixed-Point Tool.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Signed Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Examples

If **Integration period** is `3` and the input signal is a ramp (`1`, `2`, `3`, `4`,...), then some of the sums that form the output of this block are as follows:

- 0+0+1 = 1
- 0+1+2 = 3
- 1+2+3 = 6
- 2+3+4 = 9
- 3+4+5 = 12
- 4+5+6 = 15
- etc.

The zeros in the first few sums represent initial conditions. With the **Input processing** parameter set to `Elements as channels`, then the values 1, 3, 6,... are successive values of the scalar output signal. With the **Input processing** parameter set to `Columns as channels`, the values 1, 3, 6,... are organized into output frames that have the same vector length as the input signal.

See Also

Integrate and Dump, Discrete-Time Integrator (Simulink documentation)

Alphabetical List

comm.ACPR

Purpose

Adjacent Channel Power Ratio measurements

Description

The ACPR System object measures adjacent channel power ratio (ACPR) of an input signal.

Construction

H = comm.ACPR creates a System object, H, that measures adjacent channel power ratio (ACPR) of an input signal.

H = comm.ACPR(Name,Value) creates 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

NormalizedFrequency

Assume normalized frequency values

Specify whether the frequency values are normalized. If you set this property to true, the object assumes that frequency values are normalized (in the [-1 1] range). The default is false. If you set this property to false, the object assumes that frequency values are measured in Hertz.

SampleRate

Sample rate of input signal

Specify the sample rate of the input signal, in samples per second, as a double-precision, positive scalar. The default is 1e6 samples per second. This property applies when you set the NormalizedFrequency property to false.

MainChannelFrequency

Main channel center frequency

Specify the main channel center frequency as a double-precision scalar. The default is 0 Hz.

When you set the NormalizedFrequency property to true, you must specify the center frequency as a normalized value between -1 and 1.

When you set the NormalizedFrequency property to false, you must specify the center frequency in Hertz. The object measures the main channel power in the bandwidth that you specify in the MainMeasurementBandwidth property. This measurement is taken at the center of the frequency that you specify in the MainMeasurementBandwidth property.

MainMeasurementBandwidth

Main channel measurement bandwidth

Specify the main channel measurement bandwidth as a double-precision, positive scalar. The default is 50e3 Hz.

When you set the NormalizedFrequency property to true, you must specify the measurement bandwidth as a normalized value between 0 and 1.

When you set the NormalizedFrequency property to false, you must specify the measurement bandwidth in Hertz. The object measures the main channel power in the bandwidth that you specify in the MainMeasurementBandwidth property. This measurement is taken at the center of the frequency that you specify in the MainChannelFrequency property.

AdjacentChannelOffset

Adjacent channel frequency offsets

Specify the adjacent channel offsets as a double-precision scalar or as a row vector comprising frequencies that define the location of adjacent channels of interest. The default is [-100e3 100e3] Hz.

When you set the NormalizedFrequency property to true, you must specify normalized frequency offset values between -1 and 1. When you set the NormalizedFrequency property to false, you must specify frequency offset values in Hertz. The offset values indicate the distance between the main channel center frequency and adjacent channel center frequencies. Positive offsets indicate adjacent channels to the right of the main channel center frequency. Negative offsets indicate adjacent channels to the left of the main channel center frequency.

AdjacentMeasurementBandwidth

Adjacent channel measurement bandwidths

Specify the measurement bandwidth for each adjacent channel. The default is the scalar, `50e3`. The object assumes that each adjacent bandwidth is centered at the frequency defined by the corresponding frequency offset. You define this offset in the `AdjacentChannelOffset` property. Set this property to a double-precision scalar or row vector of length equal to the number of specified offsets in the `AdjacentChannelOffset` property.

When you set this property to a scalar, the object obtains all adjacent channel power measurements within equal measurement bandwidths. When you set the `NormalizedFrequency` property to `true`, you must specify normalized bandwidth values between `0` and `1`. When you set the `NormalizedFrequency` property to `false`, you must specify the adjacent channel bandwidth values in Hertz.

MeasurementFilterSource

Source of the measurement filter

Specify the measurement filter source as one of `None | Property`. The default is `None`. When you set this property to `None` the object does not apply filtering to obtain ACPR measurements. When you set this property to `Property`, the object applies a measurement filter to the main channel before measuring the average power. Each of the adjacent channel bands also receives a measurement filter . In this case, you specify the measurement filter coefficients in the `MeasurementFilter` property.

MeasurementFilter

Measurement filter coefficients

Specify the measurement filter coefficients as a double-precision row vector containing the coefficients of an FIR filter in descending order of powers of z. Center the response of the filter at DC. The ACPR object automatically shifts and applies the filter response at each of the main and adjacent channel center frequencies before obtaining the average power measurements. The internal filter states persist and clear only when you call the reset method. This property applies when you set the `MeasurementFilter` property to `Property`. The default is `1`, which is an all-pass filter that has no effect on the measurements.

SpectralEstimation

Spectral estimation control

Specify the spectral estimation control as one of `Auto | Specify frequency resolution | Specify window parameters`. The default is `Auto`.

When you set this property to `Auto`, the object obtains power measurements with a Welch spectral estimator with zero-percent overlap, a Hamming window, and a segment length equal to the length of the input data vector. In this setting, the spectral estimator set should achieve the maximum frequency resolution attainable with the input data length.

When you set this property to `Specify frequency resolution`, you specify the desired spectral frequency resolution, in normalized units or in Hertz, using the `FrequencyResolution` property. In this setting, the object uses the value in the `FrequencyResolution` property to automatically compute the size of the spectral estimator data window.

When you set this property to `Specify window parameters`, several spectral estimator properties become available so that you can control the Welch spectral estimation settings. These properties are: `SegmentLength`, `OverlapPercentage`, `Window`, and `SidelobeAttenuation`. Sidelobe attenuation applies only when you set the `Window` property to `Chebyshev`.

When you set the this property to Specify window parameters, the FrequencyResolution property does not apply, and you control the resolution using the above properties.

SegmentLength

Segment length

Specify the segment length, in samples, for the spectral estimator as a numeric, positive, integer scalar. The default is 64. The length of the segment allows you to make tradeoffs between frequency resolution and variance in the spectral estimates. A long segment length results in better resolution. A short segment length results in more averaging and a decrease in variance. This property applies when you set the SpectralEstimation property to Specify window parameters.

OverlapPercentage

Overlap percentage

Specify the percentage of overlap between each segment in the spectral estimator as a double-precision scalar in the [0 100] interval. This property applies when you set the SpectralEstimation property to Specify window parameters. The default is 0 percent.

Window

Window function

Specify a window function for the spectral estimator as one of Bartlett | Bartlett-Hanning | Blackman | Blackman-Harris | Bohman | Chebyshev | Flat Top | Hamming | Hann | Nuttall | Parzen | Rectangular | Triangular. The default is Hamming. A Hamming window has 42.5dB of sidelobe attenuation. This attenuation may mask spectral content below this value, relative to the peak spectral content. Choosing different windows allows you to make tradeoffs between resolution and sidelobe attenuation. This property applies when you set the SpectralEstimation property to Specify window parameters.

SidelobeAttenuation

Sidelobe attenuation for Chebyshev window

Specify the sidelobe attenuation, in decibels, for the Chebyshev window function as a double-precision, nonnegative scalar. The default is `100` dB. This property applies when you set the `SpectralEstimation` property to `Specify window parameters` and the `Window` property to `Chebyshev`.

FrequencyResolution

Frequency resolution

Specify the frequency resolution of the spectral estimator as a double-precision scalar. The default is `10625` Hz.

When you set the NormalizedFrequency property to `true`, you must specify the frequency resolution as a normalized value between `0` and `1`. When you set the `NormalizedFrequency` property to `false`, you must specify the frequency resolution in Hertz. The object uses the value in the `FrequencyResolution` property to calculate the size of the data window used by the spectral estimator. This property applies when you set the `SpectralEstimation` property to `Specify frequency resolution`.

FFTLength

FFT length

Specify the FFT length that the Welch spectral estimator uses as one of `Next power of 2 | Same as segment length | Custom`. The default is `Next power of 2`.

When you set this property to `Custom`, the `CustomFFTLength` property becomes available to specify the desired FFT length.

When you set this property to `Next power of 2`, the object sets the length of the FFT to the next power of 2. This length is greater than the spectral estimator segment length or `256`, whichever is greater.

When you set this property to `Same as segment length`, the object sets the length of the FFT. This length equals the spectral estimator segment length or `256`, whichever is greater.

CustomFFTLength

Custom FFT length

Specify the number of FFT points that the spectral estimator uses as a numeric, positive, integer scalar. This property applies when you set the `FFTLength` property to `Custom`. The default is `256`.

MaxHold

Max-hold setting control

Specify the maximum hold setting. The default is `false`.

When you set this property to `true`, the object compares two vectors. One vector compared is the current estimated power spectral density vector (obtained with the current input data frame). The object checks this vector against the previous maximum-hold accumulated power spectral density vector, (obtained at the previous call to the `step` method). The object stores the maximum values at each frequency bin and uses them to compute average power measurements. You clear the maximum-hold spectrum by calling the `reset` method on the object. When you set this property to `false`, the object obtains power measurements using instantaneous power spectral density estimates. This property is tunable.

PowerUnits

Power units

Specify power measurement units as one of `dBm | dBW | Watts`. The default is `dBm`.

When you set this property to `dBm`, or `dBW`, the `step` method outputs ACPR measurements in a dBc scale (adjacent channel power referenced to main channels power). If you set this property to `Watts`, the `step` method outputs ACPR measurements in a linear scale.

MainChannelPowerOutputPort

Enable main channel power measurement output

When you set this property to `true`, the `step` method outputs the main channel power measurement. The default is `false`. The main channel power is the power of the input signal measured in the band that you define with the `MainChannelFrequency` and `MainMeasurementBandwidth` properties. The `step` method returns power measurements in the units that you specify in the `PowerUnits` property.

AdjacentChannelPowerOutputPort

Enable adjacent channel power measurements output

When you set this property to `true`, the `step` method outputs a vector of adjacent channel power measurements. The default is `false`. The adjacent channel powers correspond to the input signal's power measured in the bands that you define with the `AdjacentChannelOffset` and `AdjacentMeasurementBandwidth` properties. The `step` method returns power measurements in the units that you specify in the `PowerUnits` property.

Methods

clone	Create ACPR measurement object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
reset	Reset states of ACPR measurement object
step	Adjacent Channel Power Ratio measurements

Examples

Measure ACPR of a 16-QAM signal with symbol rate of 3.84 Msps.

```
% Generate data with an alphabet size of 16 and modulate the data
x  = randi([0 16-1],5000,1);
hMod = comm.RectangularQAMModulator(16);
y = step(hMod,x);

% Usample the data by L = 8 using a rectangular pulse shape
L = 8;
yPulse = rectpulse(y,L);

% Create an ACPR measurement object and measure the modulated signal
h = comm.ACPR(...
       'SampleRate', 3.84e6*8,...
       'MainChannelFrequency', 0,...
       'MainMeasurementBandwidth', 3.84e6,...
       'AdjacentChannelOffset', [-5e6 5e6],...
       'AdjacentMeasurementBandwidth', 3.84e6,...
       'MainChannelPowerOutputPort', true,...
       'AdjacentChannelPowerOutputPort', true);
[ACPR,mainChnlPwr,adjChnlPwr] = step(h,yPulse)
```

Algorithms

Note The following conditions must be true, otherwise power measurements fall out of the Nyquist interval.

$$\left| MainChannelFreq \pm \frac{MainChannelMeasBW}{2} \right| < F\max$$

$$\left| (MainChannelFreq + AdjChannelOffset) \pm \frac{AdjChannelMeasBW}{2} \right| < F\max$$

F_{max} = Fs/2 if NormalizedFrequency = false

F_{max} = 1 if NormalizedFrequency = true

See Also

`comm.CCDF` | `comm.EVM` | `comm.MER`

comm.ACPR.clone

Purpose Create ACPR measurement object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `ACPR` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.ACPR.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn outputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `ACPR` 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.

comm.ACPR.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 ACPR measurement object

Syntax

`reset(H)`

Description

`reset(H)` resets the states of the `ACPR` object, `H`.

comm.ACPR.step

Purpose Adjacent Channel Power Ratio measurements

Syntax

```
A = step(H,X)
[A,MAINPOW] = step(H,X)
[A,ADJPOW] = step(H,X)
```

Description

`A = step(H,X)` returns a vector of the adjacent channel power ratio, `A`, measured in the input data, `X`. The measurements are at the frequency bands that you specify with the `MainChannelFrequency`, `MainMeasurementBandwidth`, `AdjacentChannelOffset`, and `AdjacentMeasurementBandwidth` properties. Input `X` must be a double precision column vector. The length of the output vector, `A`, equals the number of adjacent channels that you specify in the `AdjacentChannelOffset` property.

`[A,MAINPOW] = step(H,X)` returns the measured main channel power, `MAINPOW`, when you set the `MainChannelPowerOutputPort` property to true. The step method outputs the main channel power measured within the main channel frequency band of interest that you specify with the `MainChannelFrequency` and `MainMeasurementBandwidth` properties.

`[A,ADJPOW] = step(H,X)` returns a vector of the measured adjacent channel powers, `ADJPOW`, when you set the `AdjacentChannelPowerOutputPort` property to true. The adjacent channel powers are measured at the adjacent frequency bands of interest that you specify with the `AdjacentChannelOffset` and `AdjacentMeasurementBandwidth` properties. The length of the output vector, `ADJPOW`, equals the length of the vector that you specify in the `AdjacentChannelOffset` property. You can combine optional output arguments when you set their enabling properties. Optional outputs must be listed in the same order as the order of the enabling properties. For example,

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.

comm.AGC

Purpose

Adaptively adjust gain for constant signal-level output

Description

The `comm.AGC` System object creates an automatic gain controller (AGC) that adaptively adjusts its gain to achieve a constant signal level at the output.

Construction

`H = comm.AGC` creates an automatic gain controller (AGC) System object, `H`, that adaptively adjusts its gain to achieve a constant signal level at the output.

`H = comm.AGC(Name,Value)` creates an AGC 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

DetectorMethod

Detector method

Specify the detector method as one of `Rectifier | Square Law`. The default is `Rectifier`.

When you set the `DetectorMethod` to `Rectifier`, the AGC detector outputs a voltage value proportional to the envelope amplitude of the output signal. The detector rectifies and then averages the input signal over the update period. The AGC adjusts the gain to obtain unity voltage at the output of the detector.

When you set the `DetectorMethod` to `Square law`, the AGC detector outputs a power value that is proportional to the square of the output voltage. The detector squares and then averages the input signal over the update period. The AGC adjusts the gain to obtain unity power at the output of the detector.

LoopMethod

Loop method

Specify the loop method of the AGC as one of `Linear | Logarithmic`. The default is `Linear`.

When you set the LoopMethod to Linear, the AGC uses the direct value of the detector output to determine the gain value. Typically, a linear loop responds quickly to increases in the input signal level. However, the loop's response to decreases in the input signal level tends to be slow.

When you set the LoopMethod to Logarithmic, the AGC uses the logarithm of the detector output to determine the gain value. Logarithmic loops respond to decreases in the input signal level much more quickly than linear loops.

UpdatePeriod

Period of gain updates in samples

Specify the period of the gain updates as a double- or single-precision, real, integer-valued scalar. The default is 100.

The number of input samples must be an integer multiple of update period. Setting the period greater than 1 increases the speed of the AGC algorithm.

If you increase the update period, you may also need to increase the step size. Similarly, if you decrease the update period, you may also need to decrease the step size.

StepSize

Step size for gain updates

Specify the step size for gain updates as a double- or single-precision, real, positive scalar. The default is 0.1.

If you increase the loop gain, the AGC responds to changes at the input signal level faster. However, gain pumping also increase.

If you increase the update period, you may also need to increase the step size. Similarly, if you decrease the update period, you may also need to decrease the step size.

MaximumGain

Maximum gain in decibels

Specify the maximum gain of the AGC in decibels as a positive scalar. The default is 30.

If the signal at the input of the AGC has a very low signal level, the AGC gain may increase rapidly. Use this property to limit the gain that the AGC applies to the input signal.

Methods

clone	Create object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of automatic gain controller
step	Apply adaptive gain to input signal

Examples

Adaptively adjust the Received Signal Amplitude to Approximately 1 Volt

Modulate a QPSK signal of amplitude 4, set the received signal amplitude to approximately 1 volt using an AGC, and then plot the output.

Create a QPSK modulated signal of 10,000 symbols with an amplitude of 4.

```
d = randi([0 3], 10000, 1);
hMod = comm.PSKModulator(4, 'PhaseOffset', pi/4);
x = 4*step(hMod, d);
```

Create an AGC System object to adjust the received signal amplitude to approximately 1.

```
hAGC = comm.AGC;
y = step(hAGC, x);
```

After the AGC reaches steady state, plot the output.

```
figure;plot(x(1000:end), '*')
hold on;plot(y(1000:end), 'or'); grid on; axis square
legend('Input of AGC', 'Output of AGC')
```

Compare the Performance of an AGC with a Rectifier Detector and a Square Law Detector

Modulate a QPSK signal, adjust the received signal level using a, and then plot the AGC response as a function of time.

Create a QPSK modulated signal of 10000 symbols with an amplitude of 4.

```
d = randi([0 3], 300, 1);
hMod = comm.PSKModulator(4, 'PhaseOffset', pi/4);
x = 4*step(hMod, d);
```

Create two AGC System objects to adjust the received signal level. Use a rectifier detector and a square law detector, each with update period 10.

```
hAGC1 = comm.AGC('DetectorMethod', 'Rectifier', 'UpdatePeriod', 10);
hAGC2 = comm.AGC('DetectorMethod', 'Square law', 'UpdatePeriod', 10);
y1 = step(hAGC1, x);
y2 = step(hAGC2, x);
```

Plot AGC response as a function of time.

```
figure;plot(abs(y1), 'b')
hold on;plot(abs(y2), 'r'); grid on; axis square
xlabel('Time'); ylabel('Amplitude')
legend('Rectifier detector', 'Square law detector')
```

Plot the Effect of Step Size on AGC Performance

Create two AGC System objects to adjust the received signal level using two different step sizes with update period 10.

Create a QPSK modulated signal of 10000 symbols with an amplitude of 4.

```
d = randi([0 3], 1000, 1);
hMod = comm.PSKModulator(4, 'PhaseOffset', pi/4);
x = 4*step(hMod, d);
```

Use two AGC System objects to adjust the received signal level. Select a step size of 0.01 and 0.1. Set the update period to 10 for both cases.

```
hAGC1 = comm.AGC('StepSize', 0.01, 'UpdatePeriod', 10);
hAGC2 = comm.AGC('StepSize', 0.1, 'UpdatePeriod', 10);
y1 = step(hAGC1, x);
y2 = step(hAGC2, x);
```

Plot AGC response as a function of time.

```
figure;plot(abs(y1), 'b')
hold on;plot(abs(y2), 'r'); grid on; axis square
xlabel('Time'); ylabel('Amplitude')
legend('Step size 0.01', 'Step size 0.1')
```

Plot Effect of Maximum Gain on Burst Signals

Create a QPSK modulated burst with a length of 2000 symbols. Prepend the burst with an empty channel by concatenating an all zero vector. Add 20 dB additive white Gaussian noise.

Create a burst signal of 2,000 QPSK modulated symbols with an amplitude of 1.

```
d = randi([0 3], 2000, 1);
hMod = comm.PSKModulator(4, 'PhaseOffset', pi/4);
x = [zeros(2000, 1); step(hMod, d)];
```

```
r = awgn(x, 20);
```

Create two AGC Sytem objects to adjust the received signal level using a maximum gain of 15 dB and 5 dB with an update period of 10 and a step size of 0.1.

```
hAGC1 = comm.AGC('MaximumGain', 15, 'UpdatePeriod', 10, 'StepSize', 0.
hAGC2 = comm.AGC('MaximumGain', 5, 'UpdatePeriod', 10, 'StepSize', 0.1
y1 = step(hAGC1, x);
y2 = step(hAGC2, x);
```

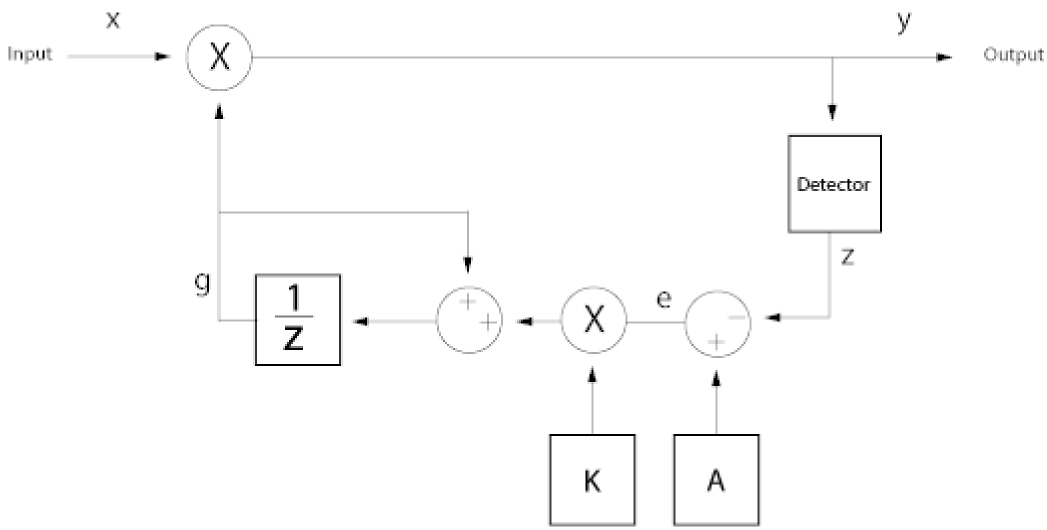
Plot the AGC response as a function of time. Limiting the maximum gain of the AGC enables the AGC to adjust faster when the packet arrives after the noise only signal.

```
figure;plot(abs(y1), 'b')
hold on;plot(abs(y2), 'r'); grid on; axis square
xlabel('Time'); ylabel('Amplitude')
legend('Maximum gain 15 dB', 'Maximum gain 12 dB')
```

Algorithms

Linear Loop AGC

In a linear loop AGC, the detector uses its output directly to generate an error signal. After applying a step size, the AGC passes the error signal to an integrator. The output of the integrator is used as the variable gain. Linear loop AGCs are limited by their decay, or slew, characteristics. In other words, they respond to input signal increases much more quickly than they respond to input signal decreases.



$$y(n) = g(n) \cdot x(n);$$
$$e(n) = A - z(m);$$
$$g(n+1) = g(n) + K \cdot e(n);$$

where

A represents the reference value, which is 1

K represents the step size

e represents the error signal

g represents the gain

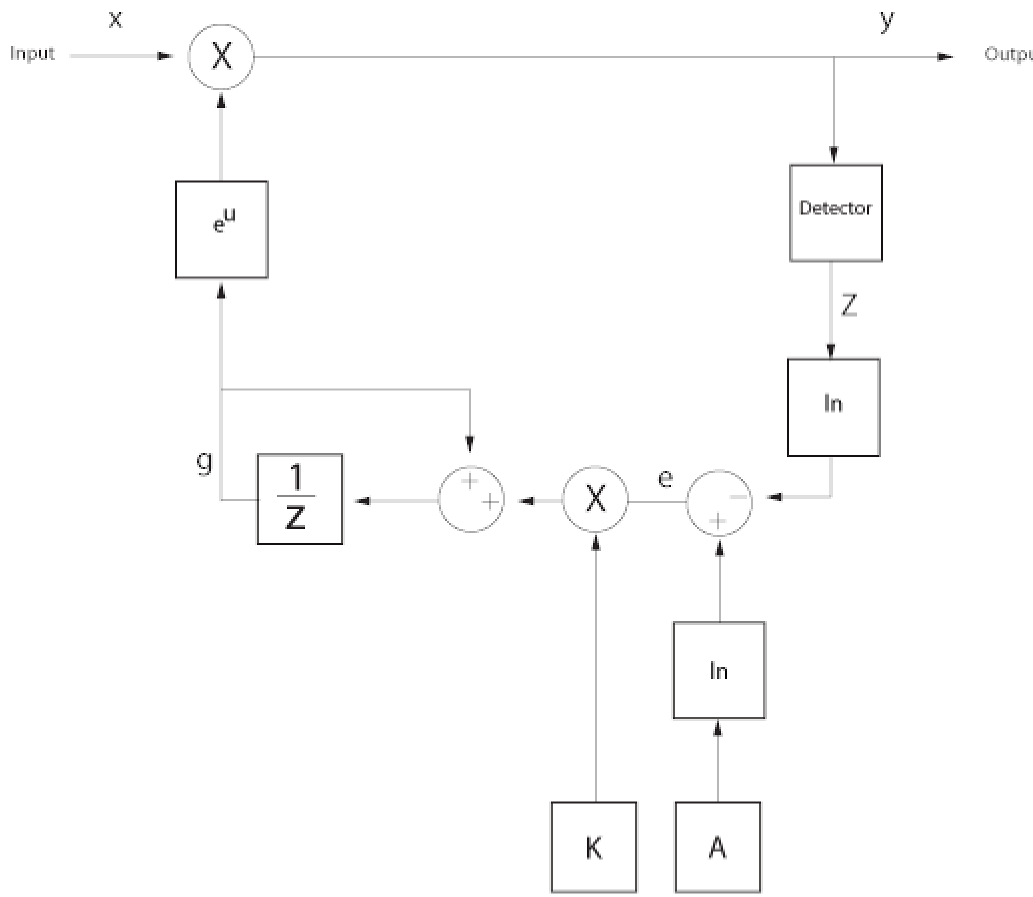
x represents the input signal

y represents the output signal

z represents the detector output

Logarithmic Loop AGC

In a logarithmic loop AGC, the logarithm of the ratio of the detector output and the reference signal represents the error signal. A logarithmic loop uses the exponential of the integrator output as the gain signal. Logarithmic-loop AGCs have the same response time to both increases or decreases to the input signal amplitude.



The logarithmic loop has longer attack and decay times. However, the gain pumping of the logarithmic loop is better than the linear loop.

$$y(n) = e^{g(n)} \cdot x(n);$$
$$e(n) = ln(A) - \ln(z(m));$$
$$g(n+1) = g(n) + K \cdot e(n);$$

where

A represents the reference value, which is 1

K represents the step size

e represents the error signal

g represents the gain

x represents the input signal

y represents the output signal

z represents the detector output

AGC Detector

Two AGC detectors are available:

Rectifier type detector

z = |y| when the detector represents a rectifier

$$z(m) = \frac{1}{N} \sum_{n=mN}^{(m+1)N-1} |y(n)|$$

where N represents the update period

Square law type detector

z = $|y|^2$ represents the square law detector

$$z(m) = \frac{1}{N} \sum_{n=mN}^{(m+1)N-1} |y(n)|^2$$

where N represents the update period

Performance Considerations

There are three performance criteria for AGCs:

- Attack time: The duration it takes the AGC to respond to an increase in the input amplitude.
- Decay time: The duration it takes the AGC to respond to a decrease in the input amplitude.
- Gain pumping: The variation in the gain value during steady-state operation.

Increasing the step size decreases the attack time and decay times, but it also increases gain pumping.

comm.AGC.clone

Purpose Create AGC object with same property values

Syntax

```
C = clone(H)
```

Description

C = clone(H) creates an AGC object C, with the same property values as H. The clone method creates a new unlocked object with uninitialized states.

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

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `AGC` 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 or hardware connections) and allows all properties and input characteristics to be changed.

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 automatic gain controller

Syntax `reset(H)`

Description `reset(H)` resets the filter states of the automatic gain controller filter object, `H`, to their initial values.

comm.AGC.step

Purpose Apply adaptive gain to input signal

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` applies an adaptive gain to the input `X`, to achieve a unity signal level at the output, `Y`. `X` must be a double or single precision column vector. The AGC determines the output signal level based on the `DetectorMethod` setting.

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

Deinterleave input symbols using algebraically derived permutation vector

Description

The `AlgebraicDeinterleaver` object restores the original ordering of a sequence that was interleaved using the `AlgebraicInterleaver` object. In typical usage, the properties of the two objects have the same values.

Construction

`H = comm.AlgebraicDeinterleaver` creates a deinterleaver System object, `H`. This object restores the original ordering of a sequence from the corresponding algebraic interleaver object.

`H = comm.AlgebraicDeinterleaver(Name,Value)` creates an Algebraic deinterleaver System 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

Method

Algebraic method to generate permutation vector

Specify the algebraic method as one of `Takeshita-Costello|Welch-Costas`. The default is `Takeshita-Costello`. The algebraic interleaver performs all computations in modulo N, where N equals the length you set in the `Length` property.

For the `Welch-Costas` method, the value of $(N+1)$ must be a prime number, where N equals the value you specify in the `Length` property. You must set the `PrimitiveElement` property to an integer, A, between 1 and N. This integer represents a primitive element of the finite field $GF(N+1)$.

For the `Takeshita-Costello` method, you must set the `Length` property to a value equal to 2^m, for any integer m. You must also set the `MultiplicativeFactor` property to an odd integer that is less than the value of the `Length` property. The `CyclicShift` property requires a nonnegative integer which is less than the value of the `Length` property. The `Takeshita-Costello` interleaver method uses a cycle vector of length N, which you specify in the Length property. The cycle vector calculation

uses the equation, $\mathrm{mod}(k \times (n-1) \times \frac{n}{2}, N)+1$, for any integer n, between 1 and N. The object creates an intermediate permutation function using the relationship, $P(c(n)) = c(n+1)$. You can shift the elements of the intermediate permutation vector to the left by the amount specified by the CyclicShift property. Doing so produces the interleaver's actual permutation vector.

Length

Number of elements in input vector

Specify the number of elements in the input as a positive, integer, scalar. When you set the Method property to Welch-Costas, then the value of Length+1 must equal a prime number. When you set the Method property to Takeshita-Costello, then the value of the Length property requires a power of two. The default is 256.

MultiplicativeFactor

Cycle vector computation factor

Specify the factor the object uses to compute the interleaver's cycle vector as a positive, integer, scalar. This property applies when you set the Method property to Takeshita-Costello. The default is 13.

CyclicShift

Amount of cyclic shift

Specify the amount by which the object shifts indices, when the object creates the final permutation vector, as a nonnegative, integer, scalar. The default is 0. This property applies when you set the Method property to Takeshita-Costello.

PrimitiveElement

Primitive element

Specify the primitive element as an element of order N in the finite field $GF(N+1)$. N is the value you specify in the `Length` property. You can express every nonzero element of $GF(N+1)$ as the value of the `PrimitiveElement` property raised to some integer power. In a Welch-Costas interleaver, the permutation maps the integer k to $\mathrm{mod}(A^k, N+1)-1$, where A represents the value of the `PrimitiveElement` property. This property applies when you set the `Method` property to `Welch-Costas`. The default is `6`.

Methods

clone	Create algebraic deinterleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Deinterleave input symbols using algebraically derived permutation vector

Examples

Interleave and deinterleave data

```
hInt = comm.AlgebraicInterleaver('Length', 16);
hDeInt = comm.AlgebraicDeinterleaver('Length', 16);
data = randi(7, 16, 1);
intData = step(hInt, data);
deIntData = step(hDeInt, intData);
[data, intData, deIntData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Algebraic Deinterleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.AlgebraicInterleaver` | `comm.BlockInterleaver`

Purpose

Create algebraic deinterleaver object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `AlgebraicDeinterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.AlgebraicDeinterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` method returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`.

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.AlgebraicDeinterleaver.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `AlgebraicDeinterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.AlgebraicDeinterleaver.step

Purpose

Deinterleave input symbols using algebraically derived permutation vector

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using an algebraic interleaver. An algebraically derived permutation vector based on the algebraic method you specify in the `Method` property forms the base of the output, `Y`. `X` must be a column vector of length specified by the `Length` property. `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `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

Permute input symbols using algebraically derived permutation vector

Description

The `AlgebraicInterleaver` object rearranges the elements of its input vector using an algebraically derived permutation.

Construction

`H = comm.AlgebraicInterleaver` creates an interleaver System object, `H`, that permutes the symbols in the input signal. This permutation is based on an algebraically derived permutation vector.

`H = comm.AlgebraicInterleaver(Name,Value)` creates an algebraic interleaver 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

Method

Algebraic method to generate permutation vector

Algebraic method to generate permutation vector

Specify the algebraic method as one of `Takeshita-Costello|` `Welch-Costas`. The default is `Takeshita-Costello`. The algebraic interleaver performs all computations in modulo N, where N is the length you set in the `Length` property.

For the `Welch-Costas` method, the value of (N+1) must be a prime number, where N is the value you specify in the `Length` property. You must set the `PrimitiveElement` property to an integer, A, between 1 and N. This integer represents a primitive element of the finite field $GF(N+1)$.

For the `Takeshita-Costello` method, you must set the `Length` property to a value equal to 2^m, for any integer m. You must also set the `MultiplicativeFactor` property to an odd integer which is less than the value of the `Length` property. In addition, you must set the `CyclicShift` property to a nonnegative integer which is less than the value of the `Length` property. The `Takeshita-Costello` interleaver method uses a cycle vector of length N, which you specify in the Length property. The cycle vector calculation uses the equation, $\mathrm{mod}(k \times (n-1) \times \frac{n}{2}, N)+1$, for any integer n, between 1 and N. The object creates an intermediate permutation function using the relationship, $P(c(n)) = c(n+1)$. You can shift the elements of the intermediate permutation vector to the left by the amount specified by the CyclicShift property. Doing so produces the actual permutation vector of the interleaver.

Length

Number of elements in input vector

Specify the number of elements in the input as a positive, integer, scalar. When you set the Method property to Welch-Costas, then the value of Length+1 must equal a prime number. When you set the Method property to Takeshita-Costello, then the value of the Length property requires a power of two. The default is 256.

MultiplicativeFactor

Cycle vector computation method

Specify the factor the object uses to compute the cycle vector for the interleaver as a positive, integer, scalar. This property applies when you set the Method property to Takeshita-Costello. The default is 13.

CyclicShift

Amount of cyclic shift

Specify the amount by which the object shifts indices, when it creates the final permutation vector, as a nonnegative, integer, scalar. This property applies when you set the Method property to Takeshita-Costello. The default is 0.

PrimitiveElement

Primitive element

Specify the primitive element as an element of order N in the finite field $GF(N+1)$. N is the value you specify in the `Length` property. You can express every nonzero element of $GF(N+1)$ as the value of the `PrimitiveElement` property raised to an integer power. In a Welch-Costas interleaver, the permutation maps the integer k to $\mathrm{mod}(A^k, N+1)-1$, where A represents the value of the `PrimitiveElement` property. This property applies when you set the `Method` property to `Welch-Costas`. The default is `6`.

Methods

clone	Create algebraic interleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Permute input symbols using an algebraically derived permutation vector

Examples

Interleave and deinterleave data

```
hInt = comm.AlgebraicInterleaver('Length', 16);
hDeInt = comm.AlgebraicDeinterleaver('Length', 16);
data = randi(7, 16, 1);
intData = step(hInt, data);
deIntData = step(hDeInt, intData);
[data, intData, deIntData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Algebraic Interleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.AlgebraicDeinterleaver`

Purpose Create algebraic interleaver object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `AlgebraicInterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.AlgebraicInterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` method returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`.

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.AlgebraicInterleaver.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `AlgebraicInterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.AlgebraicInterleaver.step

Purpose

Permute input symbols using an algebraically derived permutation vector

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The object uses an algebraically derived permutation vector, based on the algebraic method you specify in the `Method` property, to form the output. The input `X` must be a column vector of length specified by the `Length` property. `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `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

Decode convolutional code using the a posteriori probability method

Description

The `APPDecoder` object performs a posteriori probability (APP) decoding of a convolutional code.

Construction

`H = comm.APPDecoder` creates an a posteriori probability (APP) decoder System object, `H`, that decodes a convolutional code using the APP method.

`H = comm.APPDecoder(Name,Value)` creates an APP decoder 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`).

`H = comm.APPDecoder(TRELLIS,Name,Value)` creates an APP decoder object, `H`, with the `TrellisStructure` property set to `TRELLIS`, and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. The default is the result of `poly2trellis(7, [171 133], 171)`. Use the `istrellis` function to check if a structure is a valid trellis structure.

TerminationMethod

Termination method of encoded frame

Specify how the encoded frame is terminated as one of `Truncated | Terminated`. The default is `Truncated`. When you set this property to `Truncated`, the object assumes that the encoder stops after encoding the last symbol in the input frame. When you set this property to `Terminated` the object assumes that the encoder forces the trellis to end each frame in the all-zeros state by encoding additional symbols. If you use the `comm.ConvolutionalEncoder` System object to generate the encoded frame, the `TerminationMethod` values of both encoder and decoder objects must match.

Algorithm

Decoding algorithm

Specify the decoding algorithm that the object uses as one of `True APP | Max* | Max`. The default is `Max*`. When you set this property to `True APP`, the object implements true a posteriori probability decoding. When you set the property to any other value, the object uses approximations to increase the speed of the computations.

NumScalingBits

Number of scaling bits

Specify the number of bits the decoder uses to scale the input data to avoid losing precision during the computations. The default is `3`. The decoder multiplies the input by $2^{\mathrm{NumScalingBits}}$ and divides the pre-output by the same factor. This property must be a scalar integer between `0` and `8`. This property applies when you set the `Algorithm` property to `Max*`.

CodedBitLLROutputPort

Enable coded-bit LLR output

Set this property to `false` to disable the second output of the decoding step method. The default is `true`.

Methods

clone	Create APP decoder object with same property values with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of APP decoder object
step	Decode convolutional code using the a posteriori probability method

Examples

Transmit a convolutionally encoded 8-PSK-modulated bit stream through an AWGN channel, then demodulate, decode using an APP decoder, and count errors.

1

Create the Convolutional encoder, PSK Modulator, and AWGN Channel System objects.

```
noiseVar = 2e-1;
frameLength = 300;
hConEnc = comm.ConvolutionalEncoder('TerminationMethod','Truncated');
hMod = comm.PSKModulator('BitInput',true, 'PhaseOffset',0);
hChan = comm.AWGNChannel('NoiseMethod', 'Variance', ...
                'Variance',noiseVar);
```

2

Demodulate using soft-decision decoding.

```
hDemod = comm.PSKDemodulator('BitOutput',true, 'PhaseOffset',0, ..
           'DecisionMethod', 'Approximate log-likelihood ratio', .
           'Variance', noiseVar);
hAPPDec = comm.APPDecoder(...
           'TrellisStructure', poly2trellis(7, [171 133]), ...
           'Algorithm', 'True APP', 'CodedBitLLROutputPort', false
hError = comm.ErrorRate;
```

3

Decode the convolutionally encoded data. Then, convert from soft-decision to hard-decision.

```
for counter = 1:5
      data = randi([0 1],frameLength,1);
      encodedData = step(hConEnc, data);
      modSignal = step(hMod, encodedData);
      receivedSignal = step(hChan, modSignal);
      demodSignal = step(hDemod, receivedSignal);
receivedSoftBits = step(hAPPDec, zeros(frameLength,1), -demodSignal)
receivedBits = double(receivedSoftBits > 0);
     errorStats = step(hError, data, receivedBits);
end
```

The APP decoder assumes a polarization of the soft inputs that is inverse to that of the demodulator soft outputs. Therefore, you must change the sign of demodulated signal, `demodSignal`.

4

Display the error rate information.

```
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
     errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the APP Decoder block reference page. The object properties correspond to the block parameters.

See Also

`comm.ConvolutionalEncoder` | `comm.ViterbiDecoder` | `poly2trellis`

Purpose

Create APP decoder object with same property values with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `APPDecoder` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.APPDecoder.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `APPDecoder` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.APPDecoder.reset

Purpose Reset states of APP decoder object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `APPDecoder` object, `H`.

Purpose

Decode convolutional code using the a posteriori probability method

Syntax

`[LUD,LCD] = step(H,LU,LC)`

Description

`[LUD,LCD] = step(H,LU,LC)` performs APP decoding. The input `LU` is the sequence of log-likelihoods of encoder input data bits. The input `LC` is the sequence of log-likelihoods of encoded bits. Negative soft inputs are considered to be zeros and positive soft inputs are considered to be ones. The outputs, `LUD` and `LCD`, are updated versions of the input `LU` and `LC` sequences and are obtained based on information about the encoder. The inputs must be of the same data type, which can be double or single precision. The output data type is the same as the input data type. If the convolutional code uses an alphabet of 2^N symbols, the `LC` and `LCD` vector lengths are multiples of N. If the decoded data uses an alphabet of 2^K output symbols, the `LU` and `LUD` vector lengths are multiples of 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.

Purpose

Add white Gaussian noise to input signal

Description

The `AWGNChannel` object adds white Gaussian noise to a real or complex input signal. When the input uses a real-valued signal, this object adds real Gaussian noise and produces a real output signal. When the input uses a complex signal, this object adds complex Gaussian noise and produces a complex output signal.

When the inputs to the object have a variable number of channels, the `EbNo`, `EsNo`, `SNR`, `BitsPerSymbol`, `SignalPower`, `SamplesPerSymbol`, and `Variance` properties must be scalars, when applicable.

Construction

`H = comm.AWGNChannel` creates an additive white Gaussian noise (AWGN) channel System object, `H`. This object then adds white Gaussian noise to a real or complex input signal.

`H = comm.AWGNChannel(Name,Value)` creates an AWGN channel 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

NoiseMethod

Method to specify noise level

Select the method to specify the noise level as one of `Signal to noise ratio (Eb/No) | Signal to noise ratio (Es/No)| Signal to noise ratio (SNR) | Variance`. The default is `Signal to noise ratio (Eb/No)`.

EbNo

Energy per bit to noise power spectral density ratio (Eb/No)

Specify the Eb/No ratio in decibels. You can set this property to a numeric, real scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (Eb/No)`. The default is `10`. This property is tunable.

EsNo

Energy per symbol to noise power spectral density ratio (Es/No)

Specify the Es/No ratio in decibels. You can set this property to a numeric, real scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (Es/No)`. The default is `10`. This property is tunable.

SNR

Signal to noise ratio (SNR)

Specify the SNR value in decibels. You can set this property to a numeric, real scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (SNR)`. The default is `10`. This property is tunable.

BitsPerSymbol

Number of bits in one symbol

Specify the number of bits in each input symbol. You can set this property to a numeric, positive, integer scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (Eb/No)`. The default is `1` bit.

SignalPower

Input signal power in Watts

Specify the mean square power of the input signal in Watts. You can set this property to a numeric, positive, real scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (Eb/No)`, `Signal to noise ratio (Es/No)`, or `Signal to noise ratio (SNR)`. The default is `1`. The object assumes a nominal impedance of 1 Ω. This property is tunable.

SamplesPerSymbol

Number of samples per symbol

Specify the number of samples per symbol. You can set this property to a numeric, positive, integer scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (Eb/No)` or `Signal to noise ratio (Es/No)`. The default is `1`.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of `Property | Input port`. The default is `Property`. Set this property to `Input port` to specify the noise variance value using an input to the `step` method. Set this property to `Property` to specify the noise variance value using the `Variance` property. This property applies when you set the `NoiseMethod` property to `Variance`.

Variance

Noise variance

Specify the variance of the white Gaussian noise. You can set this property to a numeric, positive, real scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Variance` and the `VarianceSource` property to `Property`. The default is `1`. This property is tunable.

RandomStream

Source of random number stream

Specify the source of random number stream as one of `Global stream | mt19937ar with seed`. The default value of this property is `Global stream`.

When you set this property to `Global stream`, the object uses the current global random number stream for normally distributed random number generation.

When you set this property to `mt19937ar with seed`, the object uses the `mt19937ar` algorithm for normally distributed random number generation. In this scenario, when you call the `reset` method, the object re-initializes the random number stream to the value of the `Seed` property.

Seed

Initial seed of `mt19937ar` random number stream

Specify the initial seed of a `mt19937ar` random number generator algorithm as a double-precision, real, nonnegative integer scalar. The default value of this property is `67`.

This property applies when you set the `RandomStream` property to `mt19937ar with seed`. For each call to the `reset` method, the object re-initialize the `mt19937ar` random number stream to the `Seed` value.

Methods

clone	Create AWGN channel object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset	Reset states of the System object
step	Add white Gaussian noise to input signal

Examples

Add White Gaussian noise to an 8-PSK signal

Modulate an 8–PSK signal, add White Gaussian noise, and plot the signal to observe the effects of noise. Then, compare the results to a multi-channel response.

Create a PSK Modulator System object.

```
hMod = comm.PSKModulator;
```

Modulate the signal by calling the `step` method of the PSK modulator.

```
modData = step(hMod, randi([0 hMod.ModulationOrder-1], 1000, 1));
```

Add White Gaussian noise to the modulated signal by passing the signal through an AWGN channel.

```
hAWGN = comm.AWGNChannel('EbNo', 15, ...
'BitsPerSymbol', log2(hMod.ModulationOrder));
```

Transmit the signal through the AWGN channel by call the step method of the AWGN channel.

```
channelOutput = step(hAWGN, modData);
```

Plot the noiseless and noisy data using scatter plots to observe the effects of noise.

```
scatterplot(modData)
scatterplot(channelOutput)
```

Use a multi-channel input with greater noise by setting the `EbNo` value to `10`.

```
hAWGN.EbNo = 10;
```

Modulate the signal by calling the `step` method.

```
modData = step(hMod, randi([0 hMod.ModulationOrder-1], 1000, 1));
modData = reshape(modData, 500, 2);
```

Obtain the signal at the output of the channel by calling the `step` method.

```
channelOutput = step(hAWGN, modData);
```

Plot the channel output to see the effects that the noise has on the signal.

```
scatterplot(channelOutput(:))
```

Process two consecutive inputs with different numbers of samples

Create the AWGN System object with the EbNo ratio set to a 1–by–2 vector by typing the following syntax at the MATLAB command line.

```
h = comm.AWGNChannel('EbNo', [10 5]);
```

Process the first input signal by calling the `step` method. This object processes 128 samples per channel over two channels.

```
step(h, ones(128, 2));
```

Process the second input signal by calling the `step` method. Without being released, this object processes an additional 256 samples per channel over two channels.

```
step(h, ones(256, 2))
```

In this example, the number of samples each channels processes can change. Because the number of channels remains fixed, you can specify the `EbNo` property value as a vector.

Process two consecutive inputs with different numbers of samples and channels

This example shows how the value of a tunable property can change between two inputs of different sizes, as long as the input is a scalar.

Create an AWGN Channel System object, with the `NoiseMethod` property set to `Signal to noise ratio` and the `SNR` property value to `20`.

```
h = comm.AWGNChannel( ...
    'NoiseMethod', 'Signal to noise ratio (SNR)',...
    'SNR', 20);
```

Process the first input signal by calling the `step` method. The input has one channel.

```
step(h, ones(20, 1));
```

Process the second input signal by calling the `step` method. The input has two channels, and a different number of samples per channel.

```
step(h, ones(90, 2))
```

In this example, the number of channels can change, but the `SNR` property value must be a scalar.

Process signals with a noise variance input

This example shows the noise variance input as a scalar or a row vector, with a length equal to the number of channels of the current signal input.

Create an AWGN Channel System object, with the `NoiseMethod` property set to `Variance` and the `VarianceSource` property set to `Input port`.

```
h = comm.AWGNChannel( ...
    'NoiseMethod',    'Variance', ...
    'VarianceSource', 'Input port');
```

Process the data by calling the `step` method. The object processes 128 samples through two channels. The variance input is a 1–by–2 vector.

```
step(h, ones(128, 2), [0.1, 0.2]);
```

Process the data by calling the `step` method. The object processes 150 samples through five channels. The variance input is a scalar.

```
step(h, ones(150, 5), 1)
```

Process signals using a self-contained random stream for repeatability

This example shows how to produce the same outputs, after reset, when using a self-contained random stream.

Create an AWGN Channel System object, with the `NoiseMethod` property set to `Variance`, the and the `RandomStream` property set to `mt19937ar with seed`, and the `Seed` property set to `99`.

```
h = comm.AWGNChannel( ...
    'NoiseMethod',  'Variance', ...
    'RandomStream', 'mt19937ar with seed', ...
    'Seed',          99);
```

Process the data by calling the `step` method.

```
y1 = step(h, ones(8, 2));
```

Reset the AWGN Channel System object by calling the `reset` method. This resets the random data stream to the initial seed of 99.

```
reset(h);
```

Process the same data by calling the `step` method.

```
y2 = step(h, ones(8, 2));
```

Compare the two signals.

```
isequal(y1, y2)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the AWGN Channel block reference page. The object properties correspond to the block parameters, except for:

- The block uses a random number generator based on the V5 RANDN (Ziggurat) algorithm and an initial seed, set with the **Initial seed** parameter to initialize the random number generator. Every time the system that contains the block is run, the block generates the same sequence of random numbers. Similarly, on the object, when you set the `RandomStream` property to `mt19937ar with seed`, you can generate reproducible numbers by resetting the object.

  When you set the `RandomStream` property to `Global stream`, this object uses the MATLAB default random stream to generate random numbers. To generate reproducible numbers using this object, you can reset the MATLAB default random stream using the following code.

  ```
  reset(RandStream.getGlobalStream)
  ```

  For more information, see help for `RandStream`.

- Sometimes, the input to the `step` method is complex. In such cases, if you try to match the block and object's random generator and seed by setting the random stream of MATLAB, the random numbers do not appear in the same order.

  The object creates the random data as follows:

  $$noise = \mathrm{randn}(lengthInput,1) + 1i\times \mathrm{randn}(lengthInput,1)$$

  The block creates random data as follows:

  $$randData = \mathrm{randn}(2\times lengthInput,1)$$
  $$noise = randData(1{:}2{:}\mathrm{end}) + 1i\times randData(2{:}2{:}\mathrm{end})$$

- The **Symbol period** block parameter corresponds to the `SamplesPerSymbol` property.

- The **Variance from mask** and **Variance from port** block parameter options of the **Mode** parameter correspond to the `VarianceSource` property.

See Also

`comm.BinarySymmetricChannel`

comm.AWGNChannel.clone

Purpose Create AWGN channel object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `AWGNChannel` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.AWGNChannel.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn outputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `AWGNChannel` 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.

comm.AWGNChannel.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the `AWGNChannel` System object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `AWGNChannel` object, `H`.

If you set the `RandomStream` property of `H` to `Global stream`, the `reset` method only resets the filters. If you set `RandomStream` to `mt19937ar with seed`, the `reset` method not only resets the filters but also reinitializes the random number stream to the value of the `Seed` property.

Purpose Add white Gaussian noise to input signal

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

`Y = step(H,X)` adds white Gaussian noise to input `X` and returns the result in `Y`. Depending on the value of the `FrameBasedProcessing` property, input `X` can be a double or single precision data type scalar, vector, or matrix with real or complex values..

`Y = step(H,X,VAR)` uses input `VAR` as the variance of the white Gaussian noise. This applies when you set the `NoiseMethod` property to `Variance` and the `VarianceSource` property to `Input port`. Input `VAR` can be a numeric, positive scalar or row vector with a length equal to the number of channels.

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

Generate Barker code

Description

The `BarkerCode` object generates Barker codes to perform synchronization. *Barker codes* are subsets of PN sequences. They are short codes, with a length at most 13, which have low-correlation sidelobes. A *correlation sidelobe* is the correlation of a codeword with a time-shifted version of itself.

Construction

`H = comm.BarkerCode` creates a Barker code generator System object, `H`, that generates a Barker code of a specified length.

`H = comm.BarkerCode(Name,Value)` creates a Barker code generator 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

Length

Length of generated code

Specify the length of the Barker code as a numeric, integer scalar in the set `{1, 2, 3, 4, 5, 7, 11, 13}`. The default is `7`. The codes that the object generates for a specified length are listed in the following table:

Length	Barker code
1	[-1]
2	[-1 1]
3	[-1 -1 1]
4	[-1 -1 1 -1]
5	[-1 -1 -1 1 -1]
7	[-1 -1 -1 1 1 -1 1]
11	[-1 -1 -1 1 1 1 -1 1 1 -1 1]
13	[-1 -1 -1 -1 -1 1 1 -1 -1 1 -1 1 -1]

SamplesPerFrame

Number of output samples per frame

Specify the number of Barker code samples that the `step` method outputs as a numeric, integer scalar. The default is `1`. If you set this property to a value of M, then the `step` method outputs M samples of a Barker code sequence of length N. N represents the length of the code that you specify in the `Length` property.

OutputDataType

Data type of output

Specify the output data type as one of `double | int8`. The default is `double`.

Methods

clone	Create Barker code generator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of Barker code generator object
step	Generate Barker code

Examples

Generate 10 samples of a Barker code sequence of length 7.

```
hBCode = comm.BarkerCode('SamplesPerFrame', 10);
seq = step(hBCode)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Barker Code Generator block reference page. The object properties correspond to the block parameters, except:

- The block **Sample time** parameter does not have a corresponding property.
- The object only implements frame based outputs.

See Also

`comm.HadamardCode` | `comm.OVSFCode`

comm.BarkerCode.clone

Purpose Create Barker code generator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `BarkerCode` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.BarkerCode.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `BarkerCode` 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.

comm.BarkerCode.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 Barker code generator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `BarkerCode` object, `H`.

comm.BarkerCode.step

Purpose

Generate Barker code

Syntax

```
Y = step(H)
```

Description

`Y = step(H)` outputs a frame of the Barker code in column vector `Y`. You specify the frame length with the `SamplesPerFrame` property. The output code is in a bi-polar format with 0 and 1 mapped to 1 and -1, 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

Decode data using BCH decoder

Description

The BCHDecoder object recovers a binary message vector from a binary BCH codeword vector. For proper decoding, the codeword and message length values in this object must match the properties in the corresponding BCH Encoder block.

Construction

H = comm.BCHDecoder creates a BCH decoder System object, H, that performs BCH decoding.

H = comm.BCHDecoder(Name,Value) creates a BCH decoder 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

CodewordLength

Codeword length

Specify the codeword length of the BCH code as a double-precision, positive, integer scalar. The default is 15. The values of the CodewordLength and MessageLength properties must produce a valid narrow-sense BCH code. For a full-length BCH code the value of the this property must take the form $2^M - 1$. M is an integer, $3 \le M \le 16$, that corresponds to the degree of the primitive polynomial that you specify with PrimitivePolynomialSource and PrimitivePolynomial. If the this property is less than $2^M - 1$, the object assumes a shortened code.

MessageLength

Message length

Specify the message length as a double-precision, positive, integer scalar. The default is 5. The values of the CodewordLength and MessageLength properties must produce a valid narrow-sense BCH code.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as one of `Auto | Property`. The default is `Auto`. When you set this property to `Auto`, the object uses a primitive polynomial of degree M=`ceil(log2(CodewordLength+1))`. The result of `fliplr(de2bi(primpoly(M)))`, sets the value for this polynomial. Set this property to `Property` to specify a polynomial using the `PrimitivePolynomial` property.

PrimitivePolynomial

Primitive polynomial

Specify the primitive polynomial of order M, that defines the finite Galois field GF(2) as a double-precision, binary row vector with the coefficients of the polynomial in order of descending powers. This property applies when you set the `PrimitivePolynomialSource` property to `Property`. The default is `fliplr(de2bi(primpoly(4)))` `= [1 0 0 1 1]`, which corresponds to the polynomial x^4+x+1.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as one of `Auto | Property`. The default is `Auto`. When you set this property to `Auto`, the object chooses the generator polynomial automatically. The object calculates the generator polynomial based on the value of the `PrimitivePolynomialSource` property. When you set the `PrimitivePolynomialSource` property to `Auto` the object calculates the generator polynomial as `bchgenpoly(CodewordLength+`SL`,MessageLength+`SL`)`. When you set the PrimitivePolynomialSource property to `Property`, the object computes generator polynomial as `bchgenpoly(CodewordLength+SL,MessageLength+SL, PrimitivePolynomial)`. In both cases, SL = (2^M-1)-`CodewordLength` is the shortened length. and M is the degree of the primitive polynomial that you specify with

PrimitivePolynomialSource and PrimitivePolynomial. Set this property to Property to specify a generator polynomial using the GeneratorPolynomial property.

GeneratorPolynomial

Generator polynomial

Specify the generator polynomial as a binary, double-precision, row vector or as a binary Galois field row vector that represents the coefficients of the generator polynomial in order of descending powers. You must use CodewordLength–MessageLength+1 as the length of the generator polynomial. This property applies when you set the GeneratorPolynomialSource property to Property. The default is the result of bchgenpoly((15,5,[],'double')), which corresponds to a 15,5 code.

When you use this object to generate code, you must set the generator polynomial to a binary, double precision row vector.

CheckGeneratorPolynomial

Enable generator polynomial checking

Set this property to true to perform a generator polynomial check the first time you call the step method. The default is true. This check verifies that $x^{\text{CodewordLength}} + 1$ is divisible by the generator polynomial specified in the GeneratorPolynomial property. For larger codes, disabling the check reduces processing time. As a best practice, perform the check at least once before setting this property to false. This property applies when you set the GeneratorPolynomialSource property to Property.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as one of None | Property. The default is None. Set this property to None to disable puncturing. Set this property to Property to decode punctured codewords based on a puncture pattern vector you specify in the PuncturePattern property.

PuncturePattern

Puncture pattern vector

Specify the pattern that the object uses to puncture the encoded data as a double-precision, binary, column vector of length `CodewordLength-MessageLength`. Zeros in the puncture pattern vector indicate the position of the parity bits that the object punctures or excludes from each codeword. This property applies when you set the `PuncturePatternSource` property to `Property`. The default is `[ones(8,1); zeros(2,1)]`.

ErasuresInputPort

Enable erasures input

Set this property to `true` to specify a vector of erasures as a `step` method input. The default is `false`. The erasures vector is a double-precision or logical, binary, column vector that indicates which bits of the input codewords to erase or ignore. The length of the vector must equal the encoded data input, (that is, the length must be an integer multiple of (`CodewordLength` – number of punctures)). Values of `1` in the erasures vector correspond to erased bits in the same position of the (possibly punctured) input codewords. Set the this property to `false` to disable erasures.

NumCorrectedErrorsOutputPort

Output number of corrected errors

Set this property to `true` so that the `step` method outputs the number of corrected errors. The default is `true`.

Methods

clone	Create BCH decoder object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Decode data using a BCH decoder

Examples

Transmit a BCH-encoded, 8-DPSK-modulated bit stream through an AWGN channel, then demodulate, decode, and count errors.

```
hEnc = comm.BCHEncoder;
hMod = comm.DPSKModulator('BitInput',true);
hChan = comm.AWGNChannel(...
        'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
hDemod = comm.DPSKDemodulator('BitOutput',true);
hDec = comm.BCHDecoder;
hError = comm.ErrorRate('ComputationDelay',3);

for counter = 1:20
  data = randi([0 1], 30, 1);
  encodedData = step(hEnc, data);
  modSignal = step(hMod, encodedData);
  receivedSignal = step(hChan, modSignal);
  demodSignal = step(hDemod, receivedSignal);
  receivedBits = step(hDec, demodSignal);
  errorStats = step(hError, data, receivedBits);
end

fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the BCH Decoder block reference. The object properties correspond to the block parameters.

See Also

comm.BCHEncoder | comm.RSDecoder

Purpose

Create BCH decoder object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `BCHDecoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.BCHDecoder.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.BCHDecoder.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `BCHDecoder` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 data using a BCH decoder

Syntax

```
Y = step(H,X)
[Y,ERR] = step(H,X)
Y = step(H,X,ERASURES)
```

Description

`Y = step(H,X)` decodes input binary codewords in `X` using a (`CodewordLength`,`MessageLength`) BCH decoder with the corresponding narrow-sense generator polynomial. The step method returns the estimated message in `Y`. This syntax applies when you set the `NumCorrectedErrorsOutputPort` property to false. The input `X` must be a numeric or logical column vector. `X` must have an integer multiple of (`CodewordLength` - number of punctures) elements. Specify the number of punctures with the `PuncturePatternSource` and `PuncturePattern` properties. Each group of (`CodewordLength` - number of punctures) input elements represents one codeword to be decoded. The length of the output decoded data vector, `Y`, is an integer multiple of the message length specified in the `MessageLength` property.

`[Y,ERR] = step(H,X)` returns the number of corrected errors in output `ERR` when you set the `NumCorrectedErrorsOutputPort` property to true. A non- negative value in the ith element of the `ERR` output vector denotes the number of corrected errors in the *i*-th input codeword. A value of `-1` in the *i*-th element of the `ERR` output indicates that a decoding error occurred for the ith input codeword. A decoding error occurs when an input codeword has more errors than the error correction capability of the BCH code.

`Y = step(H,X,ERASURES)` uses `ERASURES` as the erasures pattern input when you set the `ErasuresInputPort` property to true. The object decodes the binary encoded data input, `X`, and treats as erasures the bits of the input codewords specified by the binary column vector, `ERASURES`. The length of `ERASURES` must equal the length of `X`, and its elements must be of data type double or logical. Values of `1` in the erasures vector correspond to erased bits in the same position of the (possibly punctured) input codewords.

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.

comm.BCHEncoder

Purpose

Encode data using BCH encoder

Description

The `BCHEncoder` object creates a BCH code with specified message and codeword lengths.

Construction

`H = comm.BCHEncoder` creates a BCH encoder System object, `H`, that performs BCH encoding.

`H = comm.BCHEncoder(Name,Value)` creates a BCH encoder 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

CodewordLength

Codeword length

Specify the codeword length of the BCH code as a double-precision, positive, integer scalar. The default is `15`. The values of the `CodewordLength` and `MessageLength` properties, must produce a valid narrow-sense BCH code. For a full-length BCH code the value of the this property must use the form $2^M - 1$. In this case, M is an integer, and $3 \le M \le 16$ corresponds to the degree of the primitive polynomial that you specify with the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties. If the this property is less than $2^M - 1$, the object assumes a shortened code form.

MessageLength

Message length

Specify the message length as a double-precision, positive, integer scalar. The values of the `CodewordLength` and `MessageLength` properties must produce a valid narrow-sense BCH code. The default is `5`.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as one of `Auto | Property`. The default is `Auto`. When you set this property to `Auto`, the object uses a primitive polynomial of degree M=`ceil(log2(CodewordLength+1))`. The result of `fliplr(de2bi(primpoly(M)))` sets the value for this polynomial. Set this property to `Property` to specify a polynomial using the `PrimitivePolynomial` property.

PrimitivePolynomial

Primitive polynomial

Specify the primitive polynomial of order M, that defines the finite Galois field GF(2). Use a double-precision, binary row vector with the coefficients of the polynomial in order of descending powers. This property applies when you set the `PrimitivePolynomialSource` property to `Property`. The default is `fliplr(de2bi(primpoly(4))) = [1 0 0 1 1]`, which corresponds to the polynomial $x^4 + x + 1$.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as one of `Auto | Property`. The default is `Auto`. When you set this property to `Auto`, the object chooses the generator polynomial automatically. The object computes the generator polynomial based on the value of the `PrimitivePolynomialSource` property. When you set the `PrimitivePolynomialSource` property to `Auto` the object computes the generator polynomial as `bchgenpoly(CodewordLength+`SL`,MessageLength+`SL`)`. When you set the PrimitivePolynomialSource property to 'Property', the object computes generator polynomial as `bchgenpoly(CodewordLength+SL,MessageLength+SL, PrimitivePolynomial)`. In both cases, SL = $(2^M - 1)$`-CodewordLength` is the shortened length. M

indicates the degree of the primitive polynomial that you specify with the PrimitivePolynomialSource and PrimitivePolynomial properties. Set this property to Property to specify a generator polynomial using the GeneratorPolynomial property.

GeneratorPolynomial

Generator polynomial

Specify the generator polynomial for encoding as a binary, double-precision row vector or as a binary Galois row vector that represents the coefficients of the generator polynomial in order of descending powers. The length of the generator polynomial requires a value of CodewordLength-MessageLength+1. This property applies when you set the GeneratorPolynomialSource property to Property. The default is the result of bchgenpoly(15,5,[],'double'), which corresponds to a (15,5) code.

CheckGeneratorPolynomial

Enable generator polynomial checking

Set this property to true to perform a generator polynomial check the first time you call the step method. The default is true. This check verifies that $x^{\text{CodewordLength}} + 1$ is divisible by the generator polynomial specified in the GeneratorPolynomial property. For larger codes, disabling the check reduces processing time. As a best practice, perform the check at least once before setting this property to false. This property applies when you set the GeneratorPolynomialSource property to Property.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as one of None | Property. The default is None. Set this property to None, to disable puncturing. Set this property to Property to decode punctured codewords. This decoding is based on a puncture pattern vector you specify in the `PuncturePattern` property.

PuncturePattern

Puncture pattern vector

Specify the pattern that the object uses to puncture the encoded data. Use a double-precision, binary, column vector of length `CodewordLength-MessageLength`. Zeros in the puncture pattern vector indicate the position of the parity bits that the object punctures or excludes from each codeword. This property applies when you set the `PuncturePatternSource` property to `Property`. The default is `[ones(8,1); zeros(2,1)]`.

Methods

clone	Create BCH encoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Encode data using a BCH encoder

Examples

Transmit a BCH-encoded, 8-DPSK-modulated bit stream through an AWGN channel, then demodulate, decode, and count errors.

```
hEnc = comm.BCHEncoder;
hMod = comm.DPSKModulator('BitInput',true);
hChan = comm.AWGNChannel(...
        'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
hDemod = comm.DPSKDemodulator('BitOutput',true);
```

```
hDec = comm.BCHDecoder;
hError = comm.ErrorRate('ComputationDelay',3);

for counter = 1:20
  data = randi([0 1], 30, 1);
  encodedData = step(hEnc, data);
  modSignal = step(hMod, encodedData);
  receivedSignal = step(hChan, modSignal);
  demodSignal = step(hDemod, receivedSignal);
  receivedBits = step(hDec, demodSignal);
  errorStats = step(hError, data, receivedBits);
end

fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the BCH Encoder block reference page. The object properties correspond to the block parameters.

See Also

`comm.BCHDecoder` | `comm.RSEncoder`

Purpose

Create BCH encoder object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `BCHEncoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.BCHEncoder.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.BCHEncoder.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `BCHEncoder` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.BCHEncoder.step

Purpose Encode data using a BCH encoder

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` encodes input binary data, `X`, using a (`CodewordLength`,`MessageLength`) BCH encoder with the corresponding narrow-sense generator polynomial and returns the result in vector `Y`. Input `X` must be a numeric or logical column vector with length equal to an integer multiple of the message length stored in the `MessageLength` property. A group of `MessageLength` input elements represents one message word to be encoded. The length of the encoded data output vector, `Y`, is an integer multiple of (`CodewordLength` - number of punctures). You specify the number of punctures with the `PuncturePatternSource` and `PuncturePattern` properties.

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 vector of bits to vector of integers

Description

The `BitToInteger` object maps groups of bits in the input vector to integers in the output vector.

Construction

`H = comm.BitToInteger` creates a bit-to-integer converter System object, `H`, that maps a vector of bits to a corresponding vector of integer values.

`H = comm.BitToInteger(Name,Value)` creates a bit-to-integer converter 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`).

`H = comm.BitToInteger(NUMBITS,Name,Value)` creates a bit-to-integer converter System object, `H` This object has the `BitsPerInteger` property set to `NUMBITS` and the other specified properties set to the specified values.

Properties

BitsPerInteger

Number of bits per integer

Specify the number of input bits that the object maps to each output integer. You can set this property to a scalar integer between `1` and `32`. The default is `3`.

MSBFirst

Assume first bit of input bit words is most significant bit

Set this property to true to indicate that the first bit of the input bit words is the most significant bit (MSB). The default is `true`. You can set this property to `false` to indicate that the first bit of the input bit words is the least significant bit (LSB).

SignedIntegerOutput

Output signed integers

Set this property to true to generate signed integer outputs. The default is `false`. You can set this property to `false` to generate unsigned integer outputs.

When you set this property to `false`, the output values are integers between `0` and $(2^N) - 1$. In this case, N is the value you specified in the `BitsPerInteger` property.

When you set this property to `true`, the output values are integers between $-(2^{(N-1)})$ and $(2^{(N-1)}) - 1$.

OutputDataType

Data type of output

Specify the output data type. The default is `Full precision`.

When you set the `SignedIntegerOutput` property to `false`, set this property as one of `Full precision | Smallest integer | Same as input | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32`.

When you set this property to `Same as input`, and the input data type is numeric or fixed-point (fi object), the output data has the same type as the input data.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in `Smallest unsigned integer` or `Full precision` mode.

When you set the `SignedIntegerOutput` property to `true`, specify the output data type as one of `Full precision | Smallest integer | double | single | int8 | int16 | int32`.

When you set this property to `Full precision`, the object determines the output data type based on the input data type. If the input data type is double or single precision, the output data has the same type as the input data. Otherwise, the property determines the output data type in the same way as when you set this property to `Smallest unsigned integer`.

Methods

clone	Create bit to integer converter object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Convert vector of bits to vector of integers

Examples

Convert randomly generated 4-bit words to integers.

```
hBitToInt = comm.BitToInteger(4);
% Generate three 4-bit words
  bitData = randi([0 1],3*hBitToInt.BitsPerInteger,1);
  intData = step(hBitToInt,bitData);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Bit To Integer Converter block reference page. The object properties correspond to the block parameters.

See Also

`comm.IntegerToBit | bi2de | bin2dec`

Purpose Create bit to integer converter object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `BitToInteger` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.BitToInteger.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `BitToInteger` 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.

comm.BitToInteger.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 vector of bits to vector of integers

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` converts binary input, `X`, to corresponding integers, `Y`. The input must be a scalar or a column vector and the data type can be numeric, `numerictype(0,1)`, or logical. The length of input `X` must be an integer multiple of the value you specify in the `BitsPerInteger` property. The object outputs a column vector with a length equal to length(`X`)/`BitsPerInteger`. When you set the `SignedIntegerOutput` property to `false`, the object maps each group of bits to an integer between `0` and ($2^{\text{BitsPerInteger}}$)-1. A group of bits contains N bits, where N is the value of the `BitsPerInteger` property. If you set the `SignedIntegerOutput` property to true, the object maps each group of BitsPerInteger bits to an integer between -($2^{(\text{BitsPerInteger-1})}$) and ($2^{(\text{BitsPerInteger-1})}$)-1.

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.

---

comm.BinarySymmetricChannel

Purpose

Introduce binary errors

Description

The BinarySymmetricChannel object introduces binary errors to the signal transmitted through this channel.

Construction

H = comm.BinarySymmetricChannel creates a binary symmetric channel System object, H, that introduces binary errors to the input signal with a prescribed probability.

H = comm.BinarySymmetricChannel(Name,Value) creates a binary symmetric channel 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

ErrorProbability

Probability of binary error

Specify the probability of a binary error as a scalar with a value between 0 and 1. The default is 0.05.

ErrorVectorOutputPort

Enable error vector output

When you set this property to true, the step method outputs an error signal, ERR. This error signal, in vector form, indicates where errors were introduced in the input signal, X. A value of 1 at the *i*-th element of ERR indicates that an error was introduced at the *i*-th element of X. Set the property to false if you do not want the ERR vector at the output of the step method. The default is true.

OutputDataType

Data type of output

Specify output data type as one of double | logical. The default is double.

Methods

clone	Create binary symmetric channel object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Introduce binary errors

Examples

Add binary errors with a probability of 0.2 to a binary input signal.

```
H = comm.BinarySymmetricChannel('ErrorProbability',0.2);
data = randi([0 1], 10, 1);
[noisyData, err] = step(H, data);
[data noisyData err]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Binary Symmetric Channel block reference page. The object properties correspond to the block parameters, except:This object uses the MATLAB default random stream to generate random numbers. The block uses a random number generator based on the V5 RANDN (Ziggurat) algorithm. An initial seed, set with the **Initial seed** parameter initializes the random number generator. For every system run that contains the block, the block generates the same sequence of random numbers. To generate reproducible numbers using this object, you can reset the MATLAB default random stream using the following code.

```
reset(RandStream.getGlobalStream)
```

For more information, see help for `RandStream`.

See Also

`comm.AWGNChannel`

Purpose

Create binary symmetric channel object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `BinarySymmetricChannel` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.BinarySymmetricChannel.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.BinarySymmetricChannel.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `BinarySymmetricChannel` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.BinarySymmetricChannel.step

Purpose

Introduce binary errors

Syntax

```
Y = step(H,X)
[Y,ERR] = step(H,X)
```

Description

`Y = step(H,X)` adds binary errors to the input signal `X` and returns the modified signal, `Y`. The input signal can be a vector or matrix with numeric, logical, or fixed-point (fi objects) data type elements. The step method output, `Y`, has the same dimensions as the input, `X`. If `X` input contains a non-binary value, *V*, the object considers it to be 1 when **abs**(*V*) > 0. This syntax applies when you set the `ErrorVectorOutputPort` property to false.

`[Y,ERR] = step(H,X)` returns the error signal vector, `ERR`. A value of `1` at the *i*-th element of `ERR` indicates that an error was introduced at the *i*-th element of `X`. The outputs, `Y` and `ERR`, have the same dimensions as the input, `X`. This syntax applies when you set the `ErrorVectorOutputPort` 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.

Purpose

Deinterleave input symbols using permutation vector

Description

The `BlockDeinterleaver` object rearranges the elements of its input vector without repeating or omitting any elements. The input can be real or complex.

Construction

`H = comm.BlockDeinterleaver` creates a block deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using the block interleaver System object.

`H = comm.BlockDeinterleaver(Name,Value)` creates object, `H`, with the specified property set to the specified value.

Properties

PermutationVector

Permutation vector

Specify the mapping used to permute the input symbol as a column vector of integers. The default is `[5;4;3;2;1]`. The mapping is a column vector of integers where the number of elements is equal to the length, *N*, of the input to the step method. Each element must be an integer, between 1 and *N*, with no repeated values.

Methods

clone	Create block deinterleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Deinterleave input symbols using permutation vector

Examples

Interleave and deinterleave data.

```
hInt = comm.BlockInterleaver([3 4 1 2]');
hDeInt = comm.BlockDeinterleaver([3 4 1 2]');
data = randi(7, 4, 1);
intData = step(hInt, data);
deIntData = step(hDeInt, intData);

% compare the original sequence, interleaved sequence, and restored s
[data, intData, deIntData]
```

Interleave and deinterleave data with random interleaver.

```
permVec = randperm(7)';  % Random permutation vector
   hInt = comm.BlockInterleaver(permVec);
   hDeInt = comm.BlockDeinterleaver(permVec);
   data = randi(9, 7, 1);
   intData = step(hInt, data);
   deIntData = step(hDeInt, intData);
   % compare the original sequence, interleaved sequence, and restored
   % sequence
   [data, intData, deIntData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General Block Deinterleaver block reference page. The object properties correspond to the block parameters.

See Also

comm.BlockInterleaver | comm.MatrixDeinterleaver

Purpose Create block deinterleaver object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `BlockDeinterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.BlockDeinterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.BlockDeinterleaver.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `BlockDeinterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.BlockDeinterleaver.step

Purpose

Deinterleave input symbols using permutation vector

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a block interleaver. The step method forms the output, `Y`, based on the mapping specified by the `PermutationVector` property as **Output**(`PermutationVector`(k))=**Input**(k), for k = 1:N, where N is the length of the permutation vector. The input `X` must be a column vector of the same length, N. The data type of `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `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

Permute input symbols using permutation vector

Description

The `BlockInterleaver` object permutes the symbols in the input signal. Internally, it uses a set of shift registers, each with its own delay value.

Construction

`H = comm.BlockInterleaver` creates a block interleaver System object, `H` This object permutes the symbols in the input signal based on a permutation vector.

`H = comm.BlockInterleaver(Name,Value)` creates object, `H`, with specified property set to the specified value.

Properties

PermutationVector

Permutation vector

Specify the mapping used to permute the input symbols as an integer column vector. The default is `[5;4;3;2;1]`. The number of elements of the permutation vector property must equal the length of the input vector. The `PermutationVector` property indicates the indices, in order, of the input elements that form the output vector. The relationship **Output**(*k*)=**Input**(`PermutationVector`(*k*)) describes this order. Each integer, *k*, must be between `1` and *N*, where *N* is the number of elements in the permutation vector. The elements in the `PermutationVector` property must be integers between `1` and *N* with no repetitions.

Methods

clone	Create block interleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Permute input symbols using a permutation vector

Examples

Interleave and deinterleave data.

```
hInt = comm.BlockInterleaver([3 4 1 2]');
hDeInt = comm.BlockDeinterleaver([3 4 1 2]');
data = randi(7, 4, 1);
intData = step(hInt, data);
deIntData = step(hDeInt, intData);

% compare the original sequence, interleaved sequence, and restored s
[data, intData, deIntData]
```

Interleave and deinterleave data with random interleaver.

```
permVec = randperm(7)';  % Random permutation vector
   hInt = comm.BlockInterleaver(permVec);
   hDeInt = comm.BlockDeinterleaver(permVec);
   data = randi(9, 7, 1);
   intData = step(hInt, data);
   deIntData = step(hDeInt, intData);
   % compare the original sequence, interleaved sequence, and restored
   % sequence
   [data, intData, deIntData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General Block Interleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.BlockDeinterleaver` | `comm.MatrixInterleaver`

Purpose Create block interleaver object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `BlockInterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.BlockInterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.BlockInterleaver.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `BlockInterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.BlockInterleaver.step

Purpose

Permute input symbols using a permutation vector

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The step method forms the output `Y`, based on the mapping defined by the `PermutationVector` property as **Output**(k)=**Input**(`PermutationVector`(k)), for k = 1:N, where N is the length of the `PermutationVector` property. The input `X` must be a column vector of length N. The data type of `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `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

Demodulate using BPSK method

Description

The BPSKDemodulator object demodulates a signal that was modulated using the binary phase shift keying method. The input is a baseband representation of the modulated signal.

Construction

H = comm.BPSKDemodulator creates a demodulator System object, H, that demodulates the input signal using the binary phase shift keying (BPSK) method.

H = comm.BPSKDemodulator(Name,Value) creates a BPSK demodulator 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).

H = comm.BPSKDemodulator(PHASE,Name,Value) creates a BPSK demodulator object, H, with the PhaseOffset property set to PHASE, and the other specified properties set to the specified values.

Properties

PhaseOffset

Phase of zeroth point of constellation

Specify the phase offset of the zeroth point of the constellation, in radians, as a finite, real scalar. The default is 0.

DecisionMethod

Demodulation decision method

Specify the decision method the object uses as one of Hard decision | Log-likelihood ratio | Approximate log-likelihood ratio. The default is Hard decision.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of Property | Input port. The default is Property. This property applies when you set the DecisionMethod property to Log-likelihood ratio or Approximate log-likelihood ratio.

Variance

Noise variance

Specify the variance of the noise as a nonzero, real scalar. The default is 1. If this value is very small (i.e., SNR is very high), log-likelihood ratio (LLR) computations can yield Inf or -Inf. This variance occurs because the LLR algorithm computes the exponential of very large or very small numbers using finite precision arithmetic. As a best practice in such cases, use approximate LLR because this option's algorithm does not compute exponentials. This property applies when you set the VarianceSource property to Property. This property is tunable.

OutputDataType

Data type of output

Specify the output data type as one of Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | logical. The default is Full precision. This property applies only when you set the DecisionMethod property to Hard decision. Thus, when you set the OutputDataType property to Full precision, and the input data type is single or double precision, the output data has the same data type as the input. If the input data is of a fixed-point type, then the output data type behaves as if you had set the OutputDataType property to Smallest unsigned integer. If you set the DecisionMethod property to Log-likelihood ratio or Approximate log-likelihood ratio, the output data type is the same as that of the input. In this case, that data type can only be single or double precision.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in Smallest unsigned integer or Full precision mode.

Fixed-Point Properties

DerotateFactorDataType

Data type of derotate factor

Specify the derotate factor data type as one of `Same word length as input | Custom`. The default is `Same word length as input`. This property applies when you set the `DecisionMethod` property to `Hard decision`. The object uses the derotate factor in the computations only when certain conditions exist. The step method input must be of a fixed-point type, and the `PhaseOffset` property must have a value that is not a multiple of $\pi/2$.

CustomDerotateFactorDataType

Fixed-point data type of derotate factor

Specify the derotate factor fixed-point type as an unscaled, `numerictype` object with a Signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `DecisionMethod` property to `Hard decision` and the `DerotateFactorDataType` property to `Custom`.

Methods

clone	Create BPSK demodulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Demodulate using BPSK method

Examples

Modulate and demodulate a signal using BPSK modulation.

```
   hMod = comm.BPSKModulator('PhaseOffset',pi/2);
   hAWGN = comm.AWGNChannel('NoiseMethod', ...
                     'Signal to noise ratio (SNR)','SNR',15);
   hDemod = comm.BPSKDemodulator('PhaseOffset',pi/2);

% Create an error rate calculator
   hError = comm.ErrorRate;
   for counter = 1:100
       % Transmit a 50-symbol frame
       data = randi([0 1],50,1);
       modSignal = step(hMod, data);
       noisySignal = step(hAWGN, modSignal);
       receivedData = step(hDemod, noisySignal);
       errorStats = step(hError, data, receivedData);
   end
   fprintf('Error rate = %f\nNumber of errors = %d\n', ...
     errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the BPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.BPSKModulator` | `comm.PSKDemodulator`

Purpose Create BPSK demodulator object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `BPSKDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.BPSKDemodulator.constellation

Purpose Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description `y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.BPSKDemodulator

Create a `comm.BPSKDemodulator` System object, and then calculate its ideal signal constellation.

Create a `comm.BPSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.BPSKDemodulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.BPSKDemodulator

Create a `comm.BPSKDemodulator` System object, and then plot the ideal signal constellation.

Create a `comm.BPSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.BPSKDemodulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.BPSKDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `BPSKDemodulator` 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.

comm.BPSKDemodulator.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Demodulate using BPSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

`Y = step(H,X)` demodulates input data, `X`, with the BPSK demodulator System object, `H`, and returns `Y`. Input `X` must be a scalar or a column vector with double or single precision data type. When you set the `DecisionMethod` property to `Hard decision`, the data type of the input can also be signed integer, or signed fixed point (fi objects).

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio` and the `VarianceSource` property to `Input port`. The data type of input `VAR` must be double or single precision.

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

Modulate using BPSK method

Description

The BPSKModulator object modulates using the binary phase shift keying method. The output is a baseband representation of the modulated signal.

Construction

H = comm.BPSKModulator creates a modulator System object, H, that modulates the input signal using the binary phase shift keying (BPSK) method.

H = comm.BPSKModulator(Name,Value) creates a BPSK modulator 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).

H = comm.BPSKModulator(PHASE,Name,Value) creates a BPSK modulator object, H. The object's PhaseOffset property is set to PHASE, and the other specified properties are set to the specified values.

Properties

PhaseOffset

Phase of zeroth point of constellation

Specify the phase offset of the zeroth point of the constellation, in radians, as a finite, real scalar. The default is 0.

OutputDataType

Data type of output

Specify the output data type as one of double | single | Custom. The default is double.

Fixed-Point Properties

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a numerictype object with a Signedness of Auto. The default is numerictype([],16). This property applies when you set the `OutputDataType` property to `Custom`.

Methods

clone	Create BPSK modulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Modulate using BPSK method

Examples

Modulate data using BPSK and visualize the data in a scatter plot.

```
% Create binary data symbols
    data = randi([0 1], 96, 1);
% Create a BPSK modulator System object
    hModulator = comm.BPSKModulator;
% Change the phase offset to pi/16
    hModulator.PhaseOffset = pi/16;
% Modulate and plot the data
    modData = step(hModulator, data);
    scatterplot(modData)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the BPSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also comm.BPSKDemodulator | comm.PSKModulator

Purpose

Create BPSK modulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `BPSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.BPSKModulator.constellation

Purpose Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description `y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.BPSKModulator

Create a `comm.BPSKModulator` System object, and then calculate its ideal signal constellation.

Create a `comm.BPSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.BPSKModulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Constellation View of Ideal Signal for comm.BPSKModulator

Create a `comm.BPSKModulator` System object, and then plot the ideal signal constellation .

Create a `comm.BPSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.BPSKModulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose

Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.BPSKModulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `BPSKModulator` 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.

comm.BPSKModulator.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Modulate using BPSK method

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` modulates input data, `X`, with the BPSK modulator System object, `H`. It returns the baseband modulated output, `Y`. The input must be a column vector of bits. The data type of the input can be numeric, logical, or unsigned fixed point of word length `1` (fi object).

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

Measure complementary cumulative distribution function

Description

The `CCDF` object measures the probability of a signal's instantaneous power to be a specified level above its average power.

Construction

`H = comm.CCDF` creates a complementary cumulative distribution function measurement (CCDF) System object, `H`, that measures the probability of a signal's instantaneous power to be a specified level above its average power.

`H = comm.CCDF(Name,Value)` creates a CCDF 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

NumPoints

Number of CCDF points

Specify the number of CCDF points that the object calculates. This property requires a numeric, positive, integer scalar. The default is `1000`. Use this property with the `MaximumPowerLimit` property to control the size of the histogram bins. The object uses these bins to estimate CCDF curves. This controls the resolution of the curves. All input channels must have the same number of CCDF points.

MaximumPowerLimit

Maximum expected input signal power

Specify the maximum expected input signal power limit for each input channel. The default is `50`. Set this property to a numeric scalar or row vector length equal to the number of input channels. When you set the this property to a scalar, the object assumes that the signals in all input channels have the same expected maximum power. When you set this property to a row vector length equal to the number of input channels, the object assumes that the *i*-th element of the vector is the maximum expected power for the signal at the *i*-th input channel. When you call the `step` method, the object displays the value of this property is in the units that you specify in the `PowerUnits` property. For each input channel, the object obtains CCDF results by integrating a histogram of instantaneous input signal powers. The object sets the bins of the histogram so that the last bin collects all power occurrences that are equal to, or greater than the power that you specify in this property. The object issues a warning if any input signal exceeds its specified maximum power limit. Use this property with the `NumPoints` property to control the size of the histogram bins that the object uses to estimate CCDF curves (such as control the resolution of the curves).

PowerUnits

Power units

Specify the power measurement units as one of `dBm | dBW | Watts`. The default is `dBm`. The `step` method outputs power measurements in the units specified in the `PowerUnits` property. When you set this property to `dBm` or `dBW`, the `step` method outputs relative power values in a dB scale. When you set this property to `Watts`, the `step` method outputs relative power values in a linear scale. When you call the `step` method, the object assumes that the units of `MaximumPowerLimit` have the same value you specified in the `PowerUnits` property.

AveragePowerOutputPort

Enable average power measurement output

When you set this property to `true`, the `step` method outputs running average power measurements. The default is `false`.

PeakPowerOutputPort

Enable peak power measurement output

When you set this property to `true`, the `step` method outputs running peak power measurements. The default is `false`.

PAPROutputPort

Enable PAPR measurement output

When you set this property to `true`, the `step` method outputs running peak-to-average-power measurements. The default is `false`.

Methods

clone	Create CCDF measurement object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
getPercentileRelativePower	Get relative power value for a given probability
getProbability	Get the probability for a given relative power value
isLocked	Locked status for input attributes and nontunable properties
plot	Plot CCDF curves
release	Allow property value and input characteristics changes
reset	Reset states of CCDF measurement object
step	Measure complementary cumulative distribution function

Examples

Obtain CCDF curves for 16-QAM and QPSK signals in AWGN.

```
hQAM = comm.RectangularQAMModulator(16);
hQPSK = comm.QPSKModulator;
hChan = comm.AWGNChannel('NoiseMethod',...
                     'Signal to noise ratio (SNR)', 'SNR', 15);
```

```
% Create a CCDF System object and request average power and peak
% power measurement outputs
   hCCDF = comm.CCDF('AveragePowerOutputPort', true, ...
                     'PeakPowerOutputPort', true);
% Modulate signals
   sQAM = step(hQAM,randi([0 16-1],20e3,1));
   sQPSK = step(hQPSK,randi([0 4-1],20e3,1));
% Pass signals through an AWGN channel
   hChan.SignalPower = 10;
   sQAMNoisy = step(hChan,sQAM);
   hChan.SignalPower = 1;
   sQPSKNoisy = step(hChan,sQPSK);
   % Obtain CCDF measurements
   [CCDFy,CCDFx,AvgPwr,PeakPwr] = step(hCCDF,[sQAMNoisy sQPSKNoisy]);
   % plot CCDF curves using the plot method of the CCDF object
   plot(hCCDF)
   legend('16-QAM','QPSK')
```

See Also

comm.ACPR | comm.EVM | comm.MER

Purpose Create CCDF measurement object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `CCDF` 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)` method returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`.

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn outputs on or off are changed.

Purpose

Get relative power value for a given probability

Syntax

`R = getPercentileRelativePower(H,P)`

Description

`R = getPercentileRelativePower(H,P)` finds the relative power values, `R`. The power of the signal of interest is above its average power by `R` dB (if PowerUnits equals 'dBW', or 'dBm') or by a factor of `R` (in linear scale if PowerUnits equals 'Watts') with a probability `P`.

The method output `R`, is a column vector with the *i*-th element corresponding to the relative power for the *i*-th input channel. The method input `P` can be a double precision scalar, or a vector with a number of elements equal to the number of input channels. If `P` is a scalar, then all the relative powers in `R` correspond to the same probability value specified in `P`. If `P` is a vector, then the *i*-th element of `R` corresponds to a power value that occurs in the *i*-th input channel, with a probability specified in the *i*-th element of `P`.

For the *i*-th input channel, this method evaluates the inverse CCDF curve at probability value $P(i)$.

Examples

Obtain CCDF curves for a unit variance AWGN signal and a dual- one signal. The AWGN signal is RPW1 dB above its average power one percent of the time, and the dual-tone signal is RPW2 dB above its average power 10 percent of the time. This example finds the values of RPW1 and RPW2.

```
n = [0:5e3-1].';
s1 = randn(5e3,1);                    % AWGN signal
s2 = sin(0.01*pi*n)+sin(0.03*pi*n);   % dual-tone signal
hCCDF = comm.CCDF;                    % create a CCDF object
step(hCCDF,[s1 s2]);                  % step the CCDF measurements
plot(hCCDF)                           % plot CCDF curves
legend('AWGN','Dual-tone')
RPW = getPercentileRelativePower(hCCDF,[1 10]);
RPW1 = RPW(1)
RPW2 = RPW(2)
```

comm.CCDF.getProbability

Purpose Get the probability for a given relative power value

Syntax

```
P = getProbability(H,R)
```

Description `P = getProbability(H,R)` finds the probability, `P`, of the power level of the signal of interest being `R` dBs (if PowerUnits equals 'dBW', or 'dBm') or Watts (if PowerUnits equals 'Watts') above its average power. `P` is a column vector with the *i*-th element corresponding to the probability value for the *i*-th input channel. Input `R` can be a double precision scalar or a vector with a number of elements equal to the number of input channels. If `R` is a scalar, then all the probability values in `P` correspond to the same relative power specified in `R`. If `R` is a vector, then the ith element of `P` contains a probability value for the *i*-th channel and for the relative power specified in the *i*-th element of `R`.

For the *i*-th input channel, this method evaluates the CCDF curve at relative power value `R`(*i*)

Examples Obtain CCDF curves for a unit variance AWGN signal and a dual- tone signal. Find the probability that the AWGN signal power is 5 dB above its average power and that the dual-tone signal power is 3 dB above its average power.

```
n = [0:5e3-1].';
s1 = randn(5e3,1);                        % AWGN signal
s2 = sin(0.01*pi*n)+sin(0.03*pi*n);       % dual-tone signal
hCCDF = comm.CCDF;
step(hCCDF,[s1 s2]);
plot(hCCDF)                               % plot CCDF curves
legend('AWGN','Dual-tone')
P = getProbability(hCCDF,[5 3])           % get probabilities
```

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `CCDF` 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.

comm.CCDF.plot

Purpose Plot CCDF curves

Syntax

```
D = plot(H)
```

Description `D = plot(H)` plots CCDF measurements in the CCDF System object, `H`. The `plot` method returns the plot handles as an output, `D`. This method plots the same number of curves as there are input channels. The `H` input can be followed by parameter-value pairs to specify additional properties of the curves. For example, `plot(H,LineWidth,2)` will create curves with line widths of 2 points.

The `comm.CCDF` System object does not support C code generation for this method.

Purpose Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description

`release(H)` Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.CCDF.reset

Purpose Reset states of CCDF measurement object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `CCDF` object, `H`.

Purpose

Measure complementary cumulative distribution function

Syntax

```
[CCDFY,CCDFX] = step(H,X)
[CCDFY,CCDFX,AVG] = step(H,X)
[CCDFY,CCDFX,PEAK] = step(H,X)
[CCDFY,CCDFX,PAPR] = step(H,X)
```

Description

`[CCDFY,CCDFX] = step(H,X)` updates CCDF, average power, and peak power measurements for input `X` using the CCDF System object, `H`. It outputs the y-axis, `CCDFY`, and x-axis, `CCDFX`, CCDF points. `X` must be a double precision, M-by-N matrix, where M is the number of time samples and N is the number of input channels. The `step` method outputs `CCDFY` as a matrix whose i-th column contains updated probability values measured from the i-th column of input matrix `X`. `CCDFY` contains the y-axis points of the CCDF curves of each channel. The `step` method outputs `CCDFX` as a matrix containing, in its i-th column, the corresponding updated instantaneous-to-average power ratios for the ith column of input matrix `X`. `CCDFX` contains the x-axis points of the CCDF curves of each channel. The object sets the number of rows in `CCDFY` and `CCDFX` equal to `NumPoints` property + 1. The probability values are percentages in the `[0 100]` interval. When you set the `PowerUnits` property to `dBW` or `dBm`, the relative powers are in dB scale. When you set the `PowerUnits` property to `Watts`, the relative powers are in linear scale. Measurements are updated each time you call the step method until you reset the object. You call the plot method to plot CCDF curves for each channel.

`[CCDFY,CCDFX,AVG] = step(H,X)` returns updated average power measurements, `AVG`, when you set the `AveragePowerOutputPort` property to true. The step method outputs `AVG` as a column vector with the ith element corresponding to an updated average power measurement for the signal available in the ith column of input matrix `X`. You specify the units for `AVG` in the `PowerUnits` property.

`[CCDFY,CCDFX,PEAK] = step(H,X)` returns updated peak power measurements, `PEAK`, when you set the `PeakPowerOutputPort` property to true. The step method outputs `PEAK` as a column vector with the ith element corresponding to an updated peak power measurement for the signal available in the ith column of input matrix X. You specify the units for PEAK in the PowerUnits property.

[CCDFY,CCDFX,PAPR] = step(H,X) returns updated peak-to-average power ratio measurements, PAPR, when you set the PAPROutputPort property to true. The step methods outputs PAPR as a column vector with the ith element corresponding to an updated peak-to-average power ratio measurement for the signal available in the ith column of input matrix X. When you set the PowerUnits property to dBW or dBm, the method outputs PAPR in a dB scale. When you set the PowerUnits property to Watts, the method outputs PAPR in a linear scale. You can combine optional output arguments when you set their enabling properties. Optional outputs must be listed in the same order as the order of the enabling properties. For example,

---

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

Display a constellation diagram for input signals

Description

The `ConstellationDiagram` System object plots constellation diagrams and provides the ability to perform EVM and MER measurements.

Construction

`H = comm.ConstellationDiagram` returns a System object, `H`, that displays real and complex-valued floating and fixed-point signals in the I/Q plane.

`H = comm.ConstellationDiagram(Name,Value, ...)` returns a Constellation Diagram System object, `H`, with each 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

Name

Caption to display on Constellation Diagram window

Specify the caption that the Constellation Diagram window displays as a string. The default value of this property is `Constellation Diagram`. This property is tunable.

SamplesPerSymbol

Number of samples used to represent a symbol

Specify the number of samples that represent a symbol. The default value of this property is `1`. When the `SamplesPerSymbol` property is greater than 1, the object downsamples and plots the input signal.

SampleOffset

Number of samples to skip before plotting points

Specify the number of samples to skip when decimating the input signal. The default value of this property is `0`. This property is tunable. This value must be a nonnegative integer less than the number of samples per symbol.

SymbolsToDisplay

The maximum number of symbols that can be displayed when input signal is long.

Always plot the latest SymbolsToDisplay symbols. The default value of this property is 256. This property is tunable.

ReferenceConstellation

The ideal constellation of the input signal

The object can display the ReferenceConstellation with its own marker. To obtain the signal quality measurement, you must set the ReferenceConstellation property to a valid value. The default value of this property is: `[0.7071+0.7071i -0.7071+0.7071i -0.7071-0.7071i 0.7070-0.7071i]`. This property is tunable.

ReferenceMarker

Specify the marker for reference display

The default value of this property is '+'. This property is tunable.

ReferenceColor

Specify the color for reference display constellation

The default value of this property is `[1 0 0]` (red). This property is tunable.

ShowReferenceConstellation

Option to turn on the reference constellation

Set this property to true to show reference constellation on the display. The default value of this property is true. This property is tunable.

Position

Scope window position in pixels

Specify the size and location of the scope window in pixels, as a four-element double vector of the form: [left bottom width height]. The default value of this property is dependent on the screen resolution, and is such that the window is positioned in the center of the screen, with a width and height of 410 and 300 pixels respectively. This property is tunable.

ShowGrid

Option to turn on grid

Set this property to `true` to turn on the grid or `false` to turn off the grid. The default value of this property is true. This property is tunable.

ShowLegend

Option to turn on legend

Set this property to true to turn on the legend. The default is false. This property is tunable.

ColorFading

Option to add color fading effect

When you set this property to true, the points in the display fade as the interval of time after they are first plotted increases. This is for animation that resembles an oscilloscope. The default value of this property is false. This property is tunable.

Title

Display title

Specify the display title as a string. The default value of this property is an empty string. This property is tunable.

XLimits

X-axis limits

Specify the x-axis limits as a two-element numeric vector: `[xmin xmax]`. The default value of this property is `[-1.375 1.375]`. This property is tunable.

YLimits

Y-axis limits

Specify the y-axis limits as a two-element numeric vector: `[ymin ymax]`. The default value of this property is `[-1.375 1.375]`. This property is tunable

XLabel

X-axis label

Specify the x-axis label as a string. The default value of this property is `In-phase Amplitude`. This property is tunable.

YLabel

Y-axis label

Specify the y-axis label as a string. The default value of this property is `Quadrature Amplitude`. This property is tunable.

MeasurementInterval

The measurement interval

When the input signal contains one sample per symbol and the reference constellation is provided, this System object can measure the signal quality in terms of EVM and MER. The measurement panel can be evoked by clicking on the Signal Quality button. This property specifies the window length for the measurement. The value of this property must be greater than one and less than or equal to the value of `SymbolesToDisplay` property. If the number of data input is less than `MeasurementInterval`, it will wait for more data before measurement can be calculated. The default value of this property is `2`. This property is tunable.

EVMNormalization

EVM normalization

Specify the normalization method that the object uses in the EVM calculation as one of `Average constellation power` or `Peak constellation power`. The default value of this property is `Average constellation power`. This property is tunable.

Methods

clone	Create scope object with same property values
hide	Hide scope window
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset internal states of the scope object
show	Make scope window visible
step	Display constellation diagram of signal in scope figure

Examples

Plot Constellation with One Sample per Symbol

Create a 16-QAM modulator, transmit data using an AWGN channel, and plot the signal constellation.

Create a Rectangular QAM Modulator System object, `hMod`, and set the modulation order to 16.

```
hMod = comm.RectangularQAMModulator('ModulationOrder', 16);
```

Transmit the modulated signal using an AWGN channel, `hAWGN`, with a signal-to-noise ratio of 20.

```
 hAWGN = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)
                                   'SNR', 20);
```

Create the constellation diagram scope.

```
hScope = comm.ConstellationDiagram('ReferenceConstellation', hMod.cons
```

Generate random data symbols as an input to the modulator.

```
d = randi([0 15], 100, 1);
```

Modulate the random data signal using the `step` method of the Rectangular QAM Modulator System object.

```
 sym = step(hMod, d);
```

Transmit the modulated signal using the `step` method of the AWGN channel System object.

```
rcv = step(hAWGN, sym);
```

Plot the transmitted signal using the `step` method of the Constellation Diagram System object

```
step(hScope, rcv)
```

Input signal has multiple samples per symbol, the ConstellationDiagram will decimate and plot one sample per symbol.

Input signal has multiple samples per symbol, the Constellation Diagram will decimate and plot one sample per symbol.

Create a Rectangular QAM Modulator System object, `hMod`, and set the modulation order to 16.

```
hMod = comm.RectangularQAMModulator('ModulationOrder', 16);
```

Create the constellation diagram scope.

```
hScope = comm.ConstellationDiagram('SamplesPerSymbol', 8,...
                  'SampleOffset', 1,...
                  'ReferenceConstellation', hMod.constellation);
```

Transmit the modulated signal using an AWGN channel, `hAWGN`, with a signal-to-noise ratio of 30.

```
 hAWGN = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',..
                                  'SNR', 30);
```

Create a normalized upsampling filter.

```
hFilDesign = fdesign.pulseshaping(8,'Raised Cosine','Nsym,Beta',8,0.5
       hFil = design(hFilDesign);
       hFil.Numerator = hFil.Numerator / max(hFil.Numerator);
       hInterp = dsp.FIRInterpolator('InterpolationFactor', hScope.Sa
                                     'Numerator', hFil.Numerator);
```

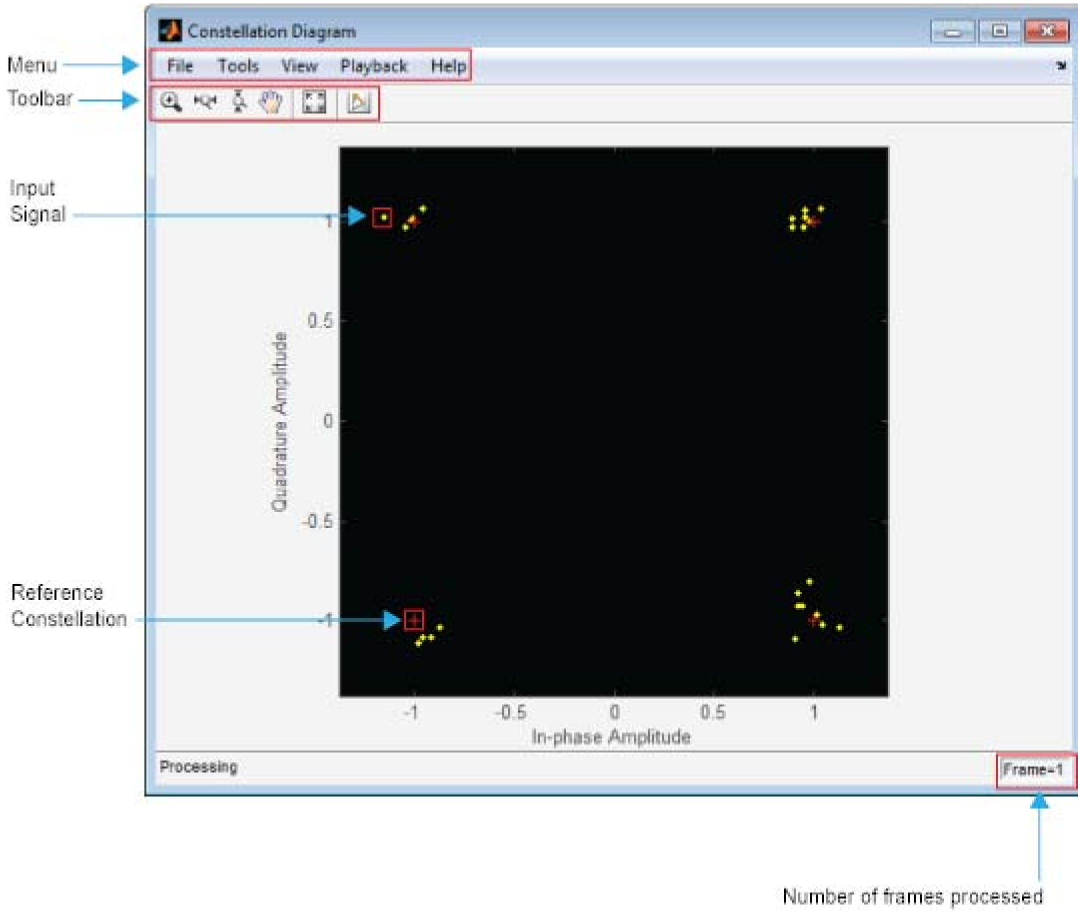
Generate a random data stream, modulate the symbols, upsample the signal and transmit using an AWGN channel. Then, display the output.

```
d = randi([0 15], 100, 1);
sym = step(hMod, d);
xmt = step(hInterp, sym);
rcv = step(hAWGN, xmt);
step(hScope, rcv)
```

Signal Display

To change the signal display settings, select **View > Configuration Properties** to bring up the Visuals—Constellation Properties dialog box. Then, modify the values for the **Samples per symbol**, **Offset**, **Symbols to display** and **Reference Constellation** parameters on the **Main** tab.

To communicate simulation data that corresponds to the current display, the scope uses the **Frames** indicator on the scope window. The following figure highlights important aspects of the Constellation Diagram window.



Menu
Toolbar
Input
Signal
Reference
Constellation
Constellation Diagram
File Tools View Playback Help
Quadrature Amplitude
In-phase Amplitude
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
Processing
Frame=1
Number of frames processed

Toolbar

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 > Pan**	N/A	When this tool is active, you can pan on the scope window. To do so, click in the center of your area of interest and drag your cursor to the left, right, up,

			or down, to move the position of the display.
	**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 after 10 Updates** — When you select this option, the scope scales the axes after 10 updates. • **Scale Axes Limits at Stop** — When you select this option, the

			scope scales the axes each time the simulation is stopped.
	**Tools > Measurements > Signal Quality**	N/A	Click this button to display Error Vector Measurement (EVM) and Modulation Error Ratio (MER) measurement results.

Measurements Panels

Measurements Panel Buttons

Each of the Measurements panels contains the following buttons that enable you to modify the appearance of the current panel.

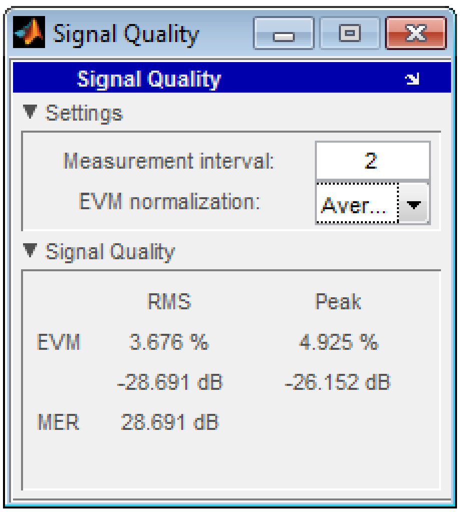
Button	Description
	Move the current panel to the top. When you are displaying more than one panel, this action moves the current panel above all the other panels.
	Collapse the current panel. When you first enable a panel, by default, it displays one or more of its panes. Click this button to hide all 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.

Button	Description
	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.

Signal Quality Panel

The Signal Quality panel displays Error Vector Measurement (EVM) and Modulation Error Ratio (MER) measurement results.



You can choose to hide or display the **Signal Quality** panel. In the Scope menu, select **Tools > Measurements > Signal Quality**.

Settings Pane

The **Settings** pane enables you to define the measurement interval and normalization method the scope uses when obtaining signal measurements.

- **Measurement interval** — Specify the duration of the EVM or MER measurement. For more information see `MeasurementInterval`.
- **EVM normalization** — For the EVM calculations, you may use one of two normalization methods: average constellation power or peak constellation power. The scope performs EVM calculations using the `comm.EVM` System object. For more information, see `comm.EVM`.

Signal Quality Pane

The **Signal Quality** pane displays the calculation results.

- **EVM** — An error vector is a vector in the I-Q plane between the ideal constellation point and the actual point at the receiver. EVM is measured in two formats: root mean square (RMS) or normalized Peak. Typically, EVM is reported in decibels. For more information, see `comm.EVM`.
- **MER** — MER is the ratio of the average power of the error vector and the average power of the transmitted signal. The scope indicates the measurement result in decibels. For more information, see `comm.MER`.

Visuals — Constellation Properties

Main Pane

Samples per symbol

Number of samples used to represent a symbol. This value must be a positive number. When the Measurements tool is on, you must set this property to `1`.

Offset (samples)

Number of samples to skip before plotting points. The offset must be a nonnegative integer value less than the value of the samples per symbol.

Symbols to display

The maximum number of symbols that can be displayed. Must be a positive integer value.

Reference constellation

The ideal constellation of the input signal. When the Measurements tool is on, the reference constellation is used to detect the ideal signal input. Therefore, this property cannot be empty when the Measurements tool is on. (When the Measurements tool is not on, this property can be empty.)

Display Pane

Show grid

Select this check box to turn on the grid.

Color fading

When you set select this check box, the points in the display fade as the interval of time after they are first plotted increases. The default value of this property is `false`. This property is tunable.

Show legend

Select this check box to display a legend for the graph.

Show reference constellation

Select this check box to display the points comprising the reference constellation.

Reference marker

Select the symbol that represents the points on the reference constellation.

Reference color

Select the color of the points on the reference constellation. Refer to the following table for the binary values and their corresponding colors.

Color	Binary Code
Black	000
Blue	001

Color	Binary Code
Green	010
Cyan	011
Red	100
Magenta	101
White	111

X-limits (Minimum)
Specify the minimum value of the x-axis.

X-limits (Maximum)
Specify the maximum value of the x-axis.

Y-limits (Minimum)
Specify the minimum value of the y-axis.

Y-limits (Maximum)
Specify the maximum value of the y-axis.

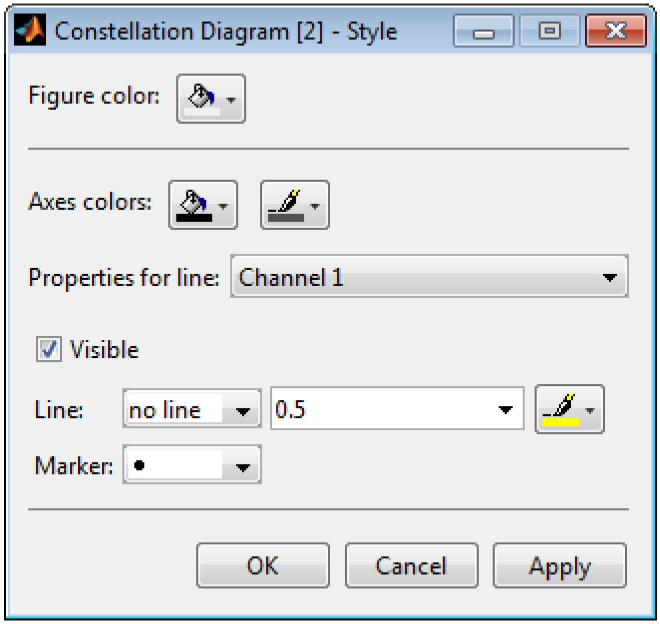
Title
Specify a label that appears above the constellation diagram plot. By default, there is no title.

X-axis label
Specify the text the scope displays along the x-axis

Y-axis label
Specify the text the scope displays along the y-axis

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 figure:

Figure color
Specify the color that you want to apply to the background of the scope figure. 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 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.

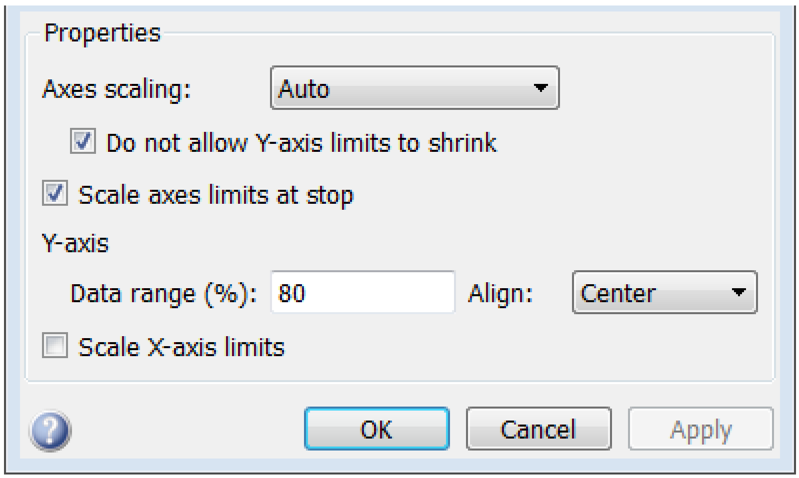
Specifier	Marker Type
none	No marker (default)
○	Circle
□	Square
×	Cross
•	Point
+	Plus sign
*	Asterisk
◇	Diamond
▽	Downward-pointing triangle
△	Upward-pointing triangle
◁	Left-pointing triangle
▷	Right-pointing triangle

Specifier	Marker Type
☆	Five-pointed star (pentagram)
✡	Six-pointed star (hexagram)

Tools: Plot Navigation Properties

Properties

The Tools—Plot Navigation Properties dialog box appears as follows.



Axes 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 figure is the active window, press **Ctrl** and **A** simultaneously.

- Auto — When you select this option, the scope scales the axes as needed, both during and after simulation. Selecting this option shows the **Do not allow Y-axis limits to shrink** check box.
- After N Updates — Selecting this option causes the scope to scale the axes after a specified number of updates. Selecting this option shows the **Number of updates** edit box.

This parameter is Tunable.

By default, this parameter is set to Manual.

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 **Axes Scaling** parameter to Manual or After N Updates, the *y*-axis limits are allowed to shrink. Tunable.

Number of updates

Specify as a positive integer the number of updates after which to scale the axes. This parameter appears only when you select After N Updates for the **Axes Scaling** parameter. Tunable.

Scale axes limits at stop

Select this check box to scale the axes limits when the simulation stops. The *y*-axis are always scaled. The *x*-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

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.

Purpose

Create scope object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a scope object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

Purpose Hide scope window

Syntax `hide(H)`

Description `hide(H)` hides the scope window associated with System object, `H`.

See Also `comm.ConstellationDiagram.show`

comm.ConstellationDiagram.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
isLocked(H)
```

Description `isLocked(H)` returns the locked state of the scope 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.

comm.ConstellationDiagram.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 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.

Algorithms

In operation, the `release` method is similar to the `mdlTerminate` function.

See Also

`comm.ConstellationDiagram.reset`

Purpose

Reset internal states of the scope object

Syntax

```
reset(H)
```

Description

`reset(H)` sets the internal states of the scope object `H` to their initial values.

You should call the `reset` method after calling the `step` method when you want to clear the 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.

Algorithms

In operation, the `reset` method is similar to a consecutive execution of the `mdlTerminate` function and the `mdlInitializeConditions` function.

See Also

`comm.ConstellationDiagram` | `comm.ConstellationDiagram.release`

comm.ConstellationDiagram.show

Purpose Make scope window visible

Syntax `show(H)`

Description `show(H)` makes the scope window associated with System object, `H`, visible.

See Also `comm.ConstellationDiagram.hide`

Purpose

Display constellation diagram of signal in scope figure

Syntax

```
step(H,X)
step(H,X1,X2,...,XN)
```

Description

`step(H,X)` displays the signal, `X`, in the scope figure.

`step(H,X1,X2,...,XN)` displays the signals `X1`, `X2`,...,`XN` in the scope figure when you set the `NumInputPorts` property to `N`. In this case, `X1`, `X2`,...,`XN` can have different data types and dimensions.

Purpose

Restore ordering of symbols using shift registers

Description

The `ConvolutionalDeinterleaver` object recovers a signal that was interleaved using the convolutional Interleaver object. The parameters in the two blocks should have the same values.

Construction

`H = comm.ConvolutionalDeinterleaver` creates a convolutional deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using the convolutional interleaver System object.

`H = comm.ConvolutionalDeinterleaver(Name,Value)` creates a convolutional deinterleaver System 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

NumRegisters

Number of internal shift registers

Specify the number of internal shift registers as a scalar, positive integer. The default is `6`.

RegisterLengthStep

Symbol capacity difference of each successive shift register

Specify the difference in symbol capacity of each successive shift register, where the last register holds zero symbols as a positive, scalar integer. The default is `2`.

InitialConditions

Initial conditions of shift registers

Specify the values that are initially stored in each shift register as a numeric scalar or vector, except the first shift register, which has zero delay. If you set this property to a scalar, then all shift registers, except the first one, store the same specified value. You can also set this property to a column vector with length equal to the value of the NumRegisters property. With this setting, the *i*-th shift register stores the *i*-th element of the specified vector. The value of the first element of this property is unimportant because the first shift register has zero delay.

Methods

clone	Create convolutional deinterleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the convolutional deinterleaver object
step	Restore ordering of symbols using shift registers

Examples

Interleave and deinterleave random data.

```
hInt = comm.ConvolutionalInterleaver('NumRegisters', 3, ...
                'RegisterLengthStep', 2, ...
                'InitialConditions', [ -1 -2 -3]');
hDeInt = comm.ConvolutionalDeinterleaver('NumRegisters', 3, ...
                'RegisterLengthStep', 2, ...
                'InitialConditions', [-1 -2 -3]');
data = (0:20)';
intrlvData = step(hInt, data);
deintrlvData = step(hDeInt, intrlvData);
```

```
% compare the original sequence, interleaved sequence and restored se
[data, intrlvData, deintrlvData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Deinterleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.ConvolutionalInterleaver` | `comm.MultiplexedInterleaver`

comm.ConvolutionalDeinterleaver.clone

Purpose

Create convolutional deinterleaver object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `ConvolutionalDeinterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.ConvolutionalDeinterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `ConvolutionalDeinterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.ConvolutionalDeinterleaver.reset

Purpose Reset states of the convolutional deinterleaver object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `ConvolutionalDeinterleaver` object, `H`.

Purpose

Restore ordering of symbols using shift registers

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a convolutional interleaver and returns `Y`. The input `X` must be a column vector. The data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The convolutional deinterleaver object uses a set of N shift registers, where N is the value specified by the `NumRegisters` property. The object sets the delay value of the k-th shift register to the product of (k-1) and `RegisterLengthStep` property value. With each new input symbol, a commutator switches to a new register and the new symbol shifts in while the oldest symbol in that register shifts out. When the commutator reaches the N-th register and the next new input occurs, it returns to the first register.

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

Convolutionally encode binary data

Description

The ConvolutionalEncoder object encodes a sequence of binary input vectors to produce a sequence of binary output vectors.

Construction

H = comm.ConvolutionalEncoder creates a System object, H, that convolutionally encodes binary data.

H = comm.ConvolutionalEncoder(Name,Value) creates a convolutional encoder 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).

H = comm.ConvolutionalEncoder(TRELLIS,Name,Value) creates a convolutional encoder object, H This object has the TrellisStructure property set to TRELLIS, and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the istrellis function to check if a structure is a valid trellis structure. The default is the result of poly2trellis(7, [171 133]).

TerminationMethod

Termination method of encoded frame

Specify how the encoded frame is terminated as one of Continuous | Truncated | Terminated. The default is Continuous. When you set this property to Continuous, the object retains the encoder states at the end of each input vector for use with the next input vector. When you set this property to Truncated, the object treats each input vector independently. The encoder states are reset at the start of each input vector. If you set the InitialStateInputPort property to false, the object resets its states to the all-zeros state. If you set the

InitialStateInputPort property to true, the object resets the states to the values you specify in the initial states step method input. When you set this property to Terminated, the object treats each input vector independently. For each input vector, the object uses extra bits to set the encoder states to all-zeros states at the end of the vector. For a rate *K*/*N* code, the *step* method outputs a vector with length $N \times {}^{(L+S)}\!/_{K}$, where S = *constraintLength*–1 (or, in the case of multiple constraint lengths, S = **sum**(*constraintLength*(*i*)–1)). L is the length of the input to the step method.

ResetInputPort

Enable encoder reset input

Set this property to true to enable an additional input to the step method. The default is false. When this additional reset input is a nonzero value, the internal states of the encoder reset to their initial conditions. This property applies when you set the TerminationMethod property to Continuous.

DelayedResetAction

Delay output reset

Set this property to true to delay resetting the object output. The default is false. When you set this property to true, the reset of the internal states of the encoder occurs after the object computes the encoded data. When you set this property to false, the reset of the internal states of the encoder occurs before the object computes the encoded data. This property applies when you set the ResetInputPort property to true.

InitialStateInputPort

Enable initial state input

Set this property to true to enable a step method input that allows the specification of the initial state of the encoder for each input vector. The default is `false`. This property applies when you set the `TerminationMethod` property to `Truncated`.

FinalStateOutputPort

Enable final state output

Set this property to true to obtain the final state of the encoder via a `step` method output. The default is `false`. This property applies when you set the `TerminationMethod` property to `Continuous` or `Truncated`.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as one of `None | Property`. The default is `None`. When you set this property to `None` the object does not apply puncturing. When you set this property to `Property`, the object punctures the code. This puncturing is based on the puncture pattern vector that you specify in the `PuncturePattern` property. This property applies when you set the `TerminationMethod` property to `Continuous` or `Truncated`.

PuncturePattern

Puncture pattern vector

Specify the puncture pattern used to puncture the encoded data as a column vector. The default is `[1; 1; 0; 1; 0; 1]`. The vector contains `1`s and `0`s, where the `0` indicates the punctured, or excluded, bits. This property applies when you set the `TerminationMethod` property to `Continuous` or `Truncated` and the `PuncturePatternSource` property to `Property`.

Methods

clone	Create convolutional encoder object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the convolutional encoder object
step	Convolutionally encode binary data

Examples

Transmit a convolutionally encoded 8-DPSK-modulated bit stream.

```
hConEnc = comm.ConvolutionalEncoder;
hMod = comm.DPSKModulator('BitInput',true);
hChan = comm.AWGNChannel('NoiseMethod', ...
                'Signal to noise ratio (SNR)',...
                'SNR',10);
hDemod = comm.DPSKDemodulator('BitOutput',true);
hDec = comm.ViterbiDecoder('InputFormat','Hard');
hError = comm.ErrorRate('ComputationDelay',3,'ReceiveDelay', 34);

for counter = 1:20
  data = randi([0 1],30,1);
  encodedData = step(hConEnc, data);
  modSignal = step(hMod, encodedData);
  receivedSignal = step(hChan, modSignal);
  demodSignal = step(hDemod, receivedSignal);
  receivedBits = step(hDec, demodSignal);
  errors = step(hError, data, receivedBits);
end
disp(errors)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Encoder block reference page. The object properties correspond to the block parameters, except:

> The operation mode **Reset on nonzero input via port** block parameter corresponds to the `ResetInputPort` property.

See Also

`comm.ViterbiDecoder` | `comm.APPDecoder`

Purpose

Create convolutional encoder object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `ConvolutionalEncoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.ConvolutionalEncoder.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.ConvolutionalEncoder.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `ConvolutionalEncoder` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the convolutional encoder object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `ConvolutionalEncoder` object, `H`.

Purpose

Convolutionally encode binary data

Syntax

```
Y = step(H,X)
Y = step(H,X,INITSTATE)
Y = step(H,X,R)
[Y,FSTATE] = step(H,X)
```

Description

`Y = step(H,X)` encodes the binary data, `X`, using the convolutional encoding that you specify in the `TrellisStructure` property. It returns the encoded data, `Y`. Both `X` and `Y` are column vectors of data type numeric, logical, or unsigned fixed point of word length `1` (fi object). When the convolutional encoder represents a rate K/N code, the length of the input vector equals $K \times L$, for some positive integer, L. The step method sets the length of the output vector, `Y`, to $L \times N$.

`Y = step(H,X,INITSTATE)` uses the initial state specified in the `INITSTATE` input when you set the `TerminationMethod` property to 'Truncated' and the `InitialStateInputPort` property to `true`. `INITSTATE` must be an integer scalar.

`Y = step(H,X,R)` resets the internal states of the encoder when you input a non-zero reset signal, `R`. `R` must be a double precision or logical scalar. This syntax applies when you set the `TerminationMethod` property to `Continuous` and the `ResetInputPort` property to true.

`[Y,FSTATE] = step(H,X)` returns the final state of the encoder in the integer scalar output `FSTATE` when you set the `FinalStateOutputPort` property to `true`. This syntax applies when you set the `TerminationMethod` property to `Continuous` or `Truncated`.

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

Permute input symbols using shift registers with same property values

Description

The `ConvolutionalInterleaver` object permutes the symbols in the input signal. Internally, this class uses a set of shift registers.

Construction

`H = comm.ConvolutionalInterleaver` creates a convolutional interleaver System object, `H`, that permutes the symbols in the input signal using a set of shift registers.

`H = comm.ConvolutionalInterleaver(Name,Value)` creates a convolutional interleaver System object, `H`. This object has 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

NumRegisters

Number of internal shift registers

Specify the number of internal shift registers as a scalar, positive integer. The default is `6`.

RegisterLengthStep

Number of additional symbols that fit in each successive shift register

Specify the number of additional symbols that fit in each successive shift register as a positive, scalar integer. The default is `2`. The first register holds zero symbols.

InitialConditions

Initial conditions of shift registers

Specify the values that are initially stored in each shift register as a numeric scalar or vector. You do not need to specify a value for the first shift register, which has zero delay. The default is `0`. The value of the first element of this property is unimportant because the first shift register has zero delay. If you set this property to a scalar, then all shift registers, except the first one, store the same specified value. If you set it to a column vector with length equal to the value of the `NumRegisters` property, then the *i*-th shift register stores the *i*-th element of the specified vector.

Methods

clone	Create convolutional interleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the convolutional interleaver object
step	Permute input symbols using shift registers

Examples

Interleave and deinterleave random data.

```
hInt = comm.ConvolutionalInterleaver('NumRegisters', 3, ...
                'RegisterLengthStep', 2, ...
                'InitialConditions', [ -1 -2 -3]');
hDeInt = comm.ConvolutionalDeinterleaver('NumRegisters', 3, ...
                'RegisterLengthStep', 2, ...
                'InitialConditions', [-1 -2 -3]');
data = (0:20)';
intrlvData = step(hInt, data);
deintrlvData = step(hDeInt, intrlvData);
% compare the original sequence, interleaved sequence and restored se
[data, intrlvData, deintrlvData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Interleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.ConvolutionalDeinterleaver` | `comm.MultiplexedInterleaver`

comm.ConvolutionalInterleaver.clone

Purpose Create convolutional interleaver object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `ConvolutionalInterleaver` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.ConvolutionalInterleaver.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `ConvolutionalInterleaver` 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.

comm.ConvolutionalInterleaver.release

Purpose Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the convolutional interleaver object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `ConvolutionalInterleaver` object, `H`.

comm.ConvolutionalInterleaver.step

Purpose

Permute input symbols using shift registers

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The input `X` must be a column vector. The data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The convolutional interleaver object uses a set of *N* shift registers, where *N* is the value specified by the `NumRegisters` property. The object sets the delay value of the *k*-th shift register to the product of (*k*-1) and the `RegisterLengthStep` property value. With each new input symbol, a commutator switches to a new register and the new symbol shifts in while the oldest symbol in that register shifts out. When the commutator reaches the *N*-th register and the next new input occurs , it returns to the first register.

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

Demodulate using CPFSK method and Viterbi algorithm

Description

The `CPFSKDemodulator` object demodulates a signal that was modulated using the continuous phase frequency shift keying method. The input is a baseband representation of the modulated signal.

Construction

`H = comm.CPFSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input continuous phase frequency shift keying (CPFSK) modulated data using the Viterbi algorithm.

`H = comm.CPFSKDemodulator(Name,Value)` creates a CPFSK demodulator 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`).

`H = comm.CPFSKDemodulator(M,Name,Value)` creates a CPFSK demodulator object, `H`. This object has the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Size of symbol alphabet

Specify the size of the symbol alphabet. The value of this property requires a power of two, real, integer scalar. The default is `4`.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer values. The default is `false`.

When you set this property to `false`, the `step` method outputs a column vector of length equal to *N*/`SamplesPerSymbol` and with elements that are integers between –(`ModulationOrder`–1) and `ModulationOrder`–1. In this case, *N*, is the length of the input signal, which indicates the number of input baseband modulated symbols.

When you set this property to `true`, the `step` method outputs a binary column vector of length equal to $P\times$ (N/`SamplesPerSymbol`), where P = **log2**(`ModulationOrder`). The output contains length-P bit words. In this scenario, the object first maps each demodulated symbol to an odd integer value, K, between –(`ModulationOrder`-1) and `ModulationOrder`–1. The object then maps K to the nonnegative integer (K+`ModulationOrder`–1)/2. Finally, the object maps each nonnegative integer to a length-P binary word, using the mapping specified in the `SymbolMapping` property.

SymbolMapping

Symbol encoding

Specify the mapping of the modulated symbols as one of `Binary | Gray`. The default is `Binary`. This property determines how the object maps each demodulated integer symbol value (in the range 0 and `ModulationOrder`–1) to a P-length bit word, where P = `ModulationOrder`(`ModulationOrder`).

When you set this property to `Binary`, the object uses a natural binary-coded ordering.

When you set this property to `Gray`, the object uses a Gray-coded ordering.

This property applies when you set the `BitOutput` property to `true`.

ModulationIndex

Modulation index

Specify the modulation index. The default is `0.5`. The value of this property can be a scalar, h, or a column vector, [h_0, h_1, ... $h_{\mathrm{H-1}}$]

where H-1 represents the length of the column vector.

When h_i varies from interval to interval, the object operates in multi-h. When the object operates in multi-h, h_i must be a rational number.

InitialPhaseOffset

Initial phase offset

Specify the initial phase offset of the input modulated waveform in radians as a real, numeric scalar. The default is `0`.

SamplesPerSymbol

Number of samples per input symbol

Specify the expected number of samples per input symbol as a positive, integer scalar. The default is `8`.

TracebackDepth

Traceback depth for Viterbi algorithm

Specify the number of trellis branches that the Viterbi algorithm uses to construct each traceback path as a positive, integer scalar. The default is `16`. The value of this property is also the value of the output delay. That value is the number of zero symbols that precede the first meaningful demodulated symbol in the output.

OutputDataType

Data type of output

Specify the output data type as one of `int8 | int16 | int32 | double`, when you set the `BitOutput` property to false. The default is `double`.

When you set the `BitOutput` property to `true`, specify the output data type as one of `logical | double`.

Methods

clone	Create CPFSK demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of CPFSK demodulator object
step	Demodulate using CPFSK method and Viterbi algorithm

Examples

Modulate and demodulate a signal using CPFSK modulation with Gray mapping and bit inputs.

```
hMod = comm.CPFSKModulator(8, 'BitInput', true, ...
                      'SymbolMapping', 'Gray');
hAWGN = comm.AWGNChannel('NoiseMethod', ...
                      'Signal to noise ratio (SNR)','SNR',0);
hDemod = comm.CPFSKDemodulator(8, 'BitOutput', true, ...
                      'SymbolMapping', 'Gray');
% Create an error rate calculator, account for the delay caused by th
% Viterbi algorithm.
delay = log2(hDemod.ModulationOrder)*hDemod.TracebackDepth;
hError = comm.ErrorRate('ReceiveDelay', delay);
for counter = 1:100
    % Transmit 100 3-bit words
    data = randi([0 1],300,1);
    modSignal = step(hMod, data);
    noisySignal = step(hAWGN, modSignal);
    receivedData = step(hDemod, noisySignal);
    errorStats = step(hError, data, receivedData);
end
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CPFSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.CPFSKModulator` | `comm.CPMModulator` | `comm.CPMDemodulator`

Purpose Create CPFSK demodulator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `CPFSKDemodulator` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.CPFSKDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `CPFSKDemodulator` 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.

comm.CPFSKDemodulator.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 CPFSK demodulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `CPFSKDemodulator` object, `H`.

comm.CPFSKDemodulator.step

Purpose Demodulate using CPFSK method and Viterbi algorithm

Syntax

```
Y = step(H,X)
```

Description `Y = step(H,X)` demodulates input data, `X`, with the CPFSK demodulator System object, `H`, and returns `Y`. Input `X` must be a double or single precision, column vector with a length equal to an integer multiple of the number of samples per symbol specified in the `SamplesPerSymbol` property. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

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

Modulate using CPFSK method

Description

The `CPFSKModulator` object modulates using the continuous phase frequency shift keying method. The output is a baseband representation of the modulated signal.

Construction

`H = comm.CPFSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the continuous phase frequency shift keying (CPFSK) modulation method.

`H = comm.CPFSKModulator(Name,Value)` creates a CPFSK modulator 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`).

`H = comm.CPFSKModulator(M,Name,Value)` creates a CPFSK modulator object, `H`. This object has the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Size of symbol alphabet

Specify the size of the symbol alphabet. The value of this property requires a power of two, real, integer scalar. The default is `4`.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`. When you set this property to `false`, the `step` method input must be a double-precision or signed integer data type column vector. This vector comprises odd integer values between –(`ModulationOrder`–1) and `ModulationOrder`–1.

When you set this property to `true`, the `step` method input must be a column vector of P-length bit words, where P = **log2**(`ModulationOrder`). The input data must have a doubleprecision or logical data type. The object maps each bit word to an integer K between 0 and ModulationOrder–1, using the mapping specified in the SymbolMapping property. The object then maps the integer K to the intermediate value $2K$–(ModulationOrder–1) and proceeds as in the case when you set the BitInput property to false.

SymbolMapping

Symbol encoding

Specify the mapping of bit inputs as one of Binary | Gray. The default is Binary. This property determines how the object maps each input P-length bit word, where P = **log2**(ModulationOrder), to an integer between 0 and ModulationOrder–1.

When you set this property to Binary, the object uses a natural binary-coded ordering.

When you set this property to Gray, the object uses a Gray-coded ordering.

This property applies when you set the BitInput property to true.

ModulationIndex

Modulation index

Specify the modulation index. The default is 0.5. The value of this property can be a scalar, h, or a column vector, $[h_0, h_1, ... h_{\mathrm{H-1}}]$

where H-1 represents the length of the column vector.

When h_{i} varies from interval to interval, the object operates in multi-h. When the object operates in multi-h, h_{i} must be a rational number.

InitialPhaseOffset

Initial phase offset

Specify the initial phase of the modulated waveform in radians as a real, numeric scalar. The default is 0.

SamplesPerSymbol

Number of samples per output symbol

Specify the upsampling factor at the output as a real, positive, integer scalar. The default is `8`. The upsampling factor is the number of output samples that the step method produces for each input sample.

OutputDataType

Data type of output

Specify output data type as one of `double | single`. The default is `double`.

Methods

clone	Create CPFSK modulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of CPFSK modulator object
step	Modulate using CPFSK method

Examples

Modulate and demodulate a signal using CPFSK modulation with Gray mapping and bit inputs.

```
hMod = comm.CPFSKModulator(8, 'BitInput', true, ...
                     'SymbolMapping', 'Gray');
hAWGN = comm.AWGNChannel('NoiseMethod', ...
                     'Signal to noise ratio (SNR)','SNR',0);
```

```
   hDemod = comm.CPFSKDemodulator(8, 'BitOutput', true, ...
                     'SymbolMapping', 'Gray');

% Create an error rate calculator, account for the delay caused by the V
   delay = log2(hDemod.ModulationOrder)*hDemod.TracebackDepth;
   hError = comm.ErrorRate('ReceiveDelay', delay);
   for counter = 1:100
       % Transmit 100 3-bit words
       data = randi([0 1],300,1);
       modSignal = step(hMod, data);
       noisySignal = step(hAWGN, modSignal);
       receivedData = step(hDemod, noisySignal);
       errorStats = step(hError, data, receivedData);
   end

   fprintf('Error rate = %f\nNumber of errors = %d\n', ...
     errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CPFSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.CPFSKDemodulator` | `comm.CPMModulator` | `comm.CPMDemodulator`

Purpose

Create CPFSK modulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `CPFSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.CPFSKModulator.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.CPFSKModulator.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `CPFSKModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.CPFSKModulator.reset

Purpose Reset states of CPFSK modulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `CPFSKModulator` object, `H`.

Purpose

Modulate using CPFSK method

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` modulates input data, `X`, with the CPFSK modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with data types double, signed integer, or logical. The length of output vector, `Y`, is equal to the number of input samples times the number of samples per symbol specified in the `SamplesPerSymbol` 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

Recover carrier phase of baseband CPM signal

Description

The CPMCarrierPhaseSynchronizer object recovers the carrier phase of the input signal using the $2P$-Power method. This feedforward method is clock aided, but not data aided. The method is suitable for systems that use certain types of baseband modulation. These types include: continuous phase modulation (CPM), minimum shift keying (MSK), continuous phase frequency shift keying (CPFSK), and Gaussian minimum shift keying (GMSK).

Construction

H = comm.CPMCarrierPhaseSynchronizer creates a CPM carrier phase synchronizer System object, H. This object recovers the carrier phase of a baseband continuous phase modulation (CPM), minimum shift keying (MSK), continuous phase frequency shift keying (CPFSK), or Gaussian minimum shift keying (GMSK) modulated signal using the 2P-power method.

H = comm.CPMCarrierPhaseSynchronizer(Name,Value) creates a CPM carrier phase synchronizer object, H This object has each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.CPMCarrierPhaseSynchronizer(HALFPOW,Name,Value) creates a CPM carrier phase synchronizer object, H. This object has the P property set to HALFPOW, and the other specified properties set to the specified values.

Properties

P

Denominator of CPM modulation index

Specify the denominator of the CPM modulation index of the input signal as a real positive scalar integer value of data type single or double. The default is 2. This property is tunable.

ObservationInterval

Number of symbols where carrier phase assumed constant

Specify the observation interval as a real positive scalar integer value of data type single or double. The default is 100.

Methods

clone	Create CPM carrier phase synchronizer object with same values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the CPM carrier phase synchronizer object
step	Recover carrier phase of baseband CPM signal

Examples

Recover carrier phase of a CPM signal using 2P-power method.

```
M = 16;
P = 2;
phOffset = 10 *pi/180;  % in radians
numSamples = 100;
% Create CPM modulator System object
  hMod = comm.CPMModulator(M, 'InitialPhaseOffset',phOffset, ...
    'BitInput',true, 'ModulationIndex',1/P, 'SamplesPerSymbol',1);
% Create CPM carrier phase synchronizer System object
  hSync = comm.CPMCarrierPhaseSynchronizer(P,...
    'ObservationInterval',numSamples);
% Generate random binary data
    data = randi([0 1],numSamples*log2(M),1);
```

```
% Modulate random data and add carrier phase
    modData = step(hMod, data);
% Recover the carrier phase
    [recSig phEst] = step(hSync, modData);
   fprintf('The carrier phase is estimated to be %g degrees.\n', phEst);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CPM Phase Recovery block reference page. The object properties correspond to the block parameters.

See Also

`comm.PSKCarrierPhaseSynchronizer` | `comm.CPMModulator`

Purpose Create CPM carrier phase synchronizer object with same values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `CPMCarrierPhaseSynchronizer` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.CPMCarrierPhaseSynchronizer.getNumInputs

Purpose Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.CPMCarrierPhaseSynchronizer.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `CPMCarrierPhaseSynchronizer` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.CPMCarrierPhaseSynchronizer.reset

Purpose Reset states of the CPM carrier phase synchronizer object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `CPMCarrierPhaseSynchronizer` object, `H`.

Purpose

Recover carrier phase of baseband CPM signal

Syntax

```
[Y,PH] = step(H,X)
```

Description

`[Y,PH] = step(H,X)` recovers the carrier phase of the input signal, `X`, and returns the phase corrected signal, `Y`, and the carrier phase estimate (in degrees), `PH`. `X` must be a complex scalar or column vector input signal of data type single or double.

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.

---

comm.CPMDemodulator

Purpose

Demodulate using CPM method and Viterbi algorithm

Description

The `CPMDemodulator` object demodulates a signal that was modulated using continuous phase modulation. The input is a baseband representation of the modulated signal.

Construction

`H = comm.CPMDemodulator` creates a demodulator System object, `H`. This object demodulates the input continuous phase modulated (CPM) data using the Viterbi algorithm.

`H = comm.CPMDemodulator(Name,Value)` creates a CPM demodulator 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`).

`H = comm.CPMDemodulator(M,Name,Value)` creates a CPM demodulator object, `H`, with the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Size of symbol alphabet

Specify the size of the symbol alphabet. The value of this property requires a power of two, real, integer scalar. The default is `4`.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer values. The default is `false`.

When you set this property to `false`, the `step` method outputs a column vector of length equal to N/`SamplesPerSymbol` and with elements that are integers between -(`ModulationOrder`-1) and `ModulationOrder`–1. Here, N, is the length of the input signal which indicates the number of input baseband modulated symbols.

When you set this property to `true`, the `step` method outputs a binary column vector of length equal to $P\times$ (N/`SamplesPerSymbol`), where P = **log2**(`ModulationOrder`). The output contains length-P bit words. In this scenario, the object first maps each demodulated symbol to an odd integer value, K, between –(`ModulationOrder`–1) and `ModulationOrder`–1. The object then maps K to the nonnegative integer (K+`ModulationOrder`–1)/2. Finally, the object maps each nonnegative integer to a length-P binary word, using the mapping specified in the `SymbolMapping` property.

SymbolMapping

Symbol encoding

Specify the mapping of the demodulated symbols as one of `Binary | Gray`. The default is `Binary`. This property determines how the object maps each demodulated integer symbol value (in the range `0` and `ModulationOrder`–1) to a P-length bit word, where P = **log2**(`ModulationOrder`).

When you set this property to `Binary`, the object uses a natural binary-coded ordering.

When you set this property to `Gray`, the object uses a Gray-coded ordering.

This property applies when you set the `BitOutput` property to true.

ModulationIndex

Modulation index

Specify the modulation index. The default is `0.5`. The value of this property can be a scalar, h, or a column vector, $[h_0, h_1, ... h_{\mathrm{H-1}}]$

where H-1 represents the length of the column vector.

When h_{i} varies from interval to interval, the object operates in multi-h. When the object operates in multi-h, h_{i} must be a rational number.

FrequencyPulse

Frequency pulse shape

Specify the type of pulse shaping that the modulator has used to smooth the phase transitions of the input modulated signal as one of `Rectangular | Raised Cosine | Spectral Raised Cosine | Gaussian | Tamed FM`. The default is `Rectangular`.

MainLobeDuration

Main lobe duration of spectral raised cosine pulse

Specify, in number of symbol intervals, the duration of the largest lobe of the spectral raised cosine pulse. This value is the value that the modulatorused to pulse-shape the input modulated signal. The default is `1`. This property requires a real, positive, integer scalar. This property applies when you set the `FrequencyPulse` property to `Spectral Raised Cosine`.

RolloffFactor

Rolloff factor of spectral raised cosine pulse

Specify the roll off factor of the spectral raised cosine pulse. This value is the value that the modulator used to pulse-shape the input modulated signal. The default is `0.2`. This property requires a real scalar between `0` and `1`. This property applies when you set the `FrequencyPulse` property to `Spectral Raised Cosine`.

BandwidthTimeProduct

Product of bandwidth and symbol time of Gaussian pulse

Specify the product of bandwidth and symbol time for the Gaussian pulse shape. This value is the value that the modulator used to pulse-shape the input modulated signal. The default is `0.3`. This property requires a real, positive scalar. This property applies when you set the `FrequencyPulse` property to `Gaussian`.

PulseLength

Pulse length

Specify the length of the frequency pulse shape in symbol intervals. The value of this property requires a real positive integer. The default is `1`.

SymbolPrehistory

Symbol prehistory

Specify the data symbols used by the modulator prior to the first call to the `step` method. The default is `1`. This property requires a scalar or vector with odd integer elements between –(`ModulationOrder`–1) and (`ModulationOrder`–1). If the value is a vector, then its length must be one less than the value in the `PulseLength` property.

InitialPhaseOffset

Initial phase offset

Specify the initial phase offset of the input modulated waveform in radians as a real, numeric scalar. The default is `0`.

SamplesPerSymbol

Number of samples per input symbol

Specify the expected number of samples per input symbol as a positive, integer scalar. The default is `8`.

TracebackDepth

Traceback depth for Viterbi algorithm

Specify the number of trellis branches that the Viterbi algorithm uses to construct each traceback path as a positive, integer scalar. The default is `16`. The value of this property is also the output delay, which is the number of zero symbols that precede the first meaningful demodulated symbol in the output.

OutputDataType

Data type of output

Specify the output data type as one of `int8 | int16 | int32 | double`, when you set the `BitOutput` property to false. When you set the `BitOutput` property to `true`, specify the output data type as one of `logical | double`. The default is `double`.

Methods

clone	Create CPM demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of CPM demodulator object
step	Demodulate using CPM method and Viterbi algorithm

Examples

Modulate and demodulate a signal using CPM modulation with Gray mapping and bit inputs.

```
    hMod = comm.CPMModulator(8, 'BitInput', true, ...
                             'SymbolMapping', 'Gray');
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                             'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.CPMDemodulator(8, 'BitOutput', true, ...
                             'SymbolMapping', 'Gray');
% Create an error rate calculator, account for the delay caused by the Vi
    delay = log2(hDemod.ModulationOrder)*hDemod.TracebackDepth;
    hError = comm.ErrorRate('ReceiveDelay', delay);
    for counter = 1:100
        % Transmit 100 3-bit words
        data = randi([0 1],300,1);
        modSignal = step(hMod, data);
        noisySignal = step(hAWGN, modSignal);
        receivedData = step(hDemod, noisySignal);
```

```
        errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CPM Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

comm.CPMModulator | comm.CPFSKDemodulator | comm.MSKDemodulator | comm.GMSKDemodulator

comm.CPMDemodulator.clone

Purpose Create CPM demodulator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `CPMDemodulator` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.CPMDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `CPMDemodulator` 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.

comm.CPMDemodulator.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 CPM demodulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `CPMDemodulator` object, `H`.

comm.CPMDemodulator.step

Purpose Demodulate using CPM method and Viterbi algorithm

Syntax

```
Y = step(H,X)
```

Description `Y = step(H,X)` demodulates input data, `X`, with the CPM demodulator System object, `H`, and returns `Y`. `X` must be a double or single precision, column vector with a length equal to an integer multiple of the number of samples per symbol specified in the `SamplesPerSymbol` property. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

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

Modulate using CPM method

Description

The CPMModulator object modulates using continuous phase modulation. The output is a baseband representation of the modulated signal.

Construction

H = comm.CPMModulator creates a modulator System object, H. This object modulates the input signal using the continuous phase modulation (CPM) method.

H = comm.CPMModulator(Name,Value) creates a CPM modulator object, H. This object has each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.CPMModulator(M,Name,Value) creates a CPM modulator object, H, with the ModulationOrder property set to M and the other specified properties set to the specified values.

Properties

ModulationOrder

Size of symbol alphabet

Specify the size of the symbol alphabet. The value of this property must be a power of two, real, integer scalar. The default is 4.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is false.

When you set this property to false, the step method input requires double-precision or signed integer data type column vector. This vector must comprise odd integer values between –(ModulationOrder–1) and ModulationOrder–1.

When you set this property to true, the step method input requires a column vector of P-length bit words, where P = **log2**(ModulationOrder). The input data must have a double-precision or logical data type. The object maps each bit word to an integer K between `0` and `ModulationOrder`–1, using the mapping specified in the `SymbolMapping` property. The object then maps the integer K to the intermediate value $2K$-(`ModulationOrder`–1) and proceeds as in the case when BitInput is false.

SymbolMapping

Symbol encoding

Specify the mapping of bit inputs as one of `Binary | Gray`. The default is `Binary`. This property determines how the object maps each input P-length bit word, where P = **log2**(`ModulationOrder`), to an integer between 0 and `ModulationOrder`–1.

When you set this property to `Binary`, the object uses a natural binary-coded ordering.

When you set this property to `Gray`, the object uses a Gray-coded ordering.

This property applies when you set the `BitInput` property to true.

ModulationIndex

Modulation index

Specify the modulation index. The default is `0.5`. The value of this property can be a scalar, h, or a column vector, $[h_0, h_1, \ldots.\ h_{\mathrm{H-1}}]$

where H-1 represents the length of the column vector.

When h_{i} varies from interval to interval, the object operates in multi-h. When the object operates in multi-h, h_{i} must be a rational number.

FrequencyPulse

Frequency pulse shape

Specify the type of pulse shaping that the modulator uses to smooth the phase transitions of the modulated signal. Choose from `Rectangular | Raised Cosine | Spectral Raised Cosine | Gaussian | Tamed FM`. The default is `Rectangular`.

MainLobeDuration

Main lobe duration of spectral raised cosine pulse

Specify, in number of symbol intervals, the duration of the largest lobe of the spectral raised cosine pulse. The default is `1`. This property requires a real, positive, integer scalar. This property applies when you set the `FrequencyPulse` property to `Spectral Raised Cosine`.

RolloffFactor

Rolloff factor of spectral raised cosine pulse

Specify the rolloff factor of the spectral raised cosine pulse. The default is `0.2`. This property requires a real scalar between `0` and `1`. This property applies when you set the `FrequencyPulse` property to `Spectral Raised Cosine`.

BandwidthTimeProduct

Product of bandwidth and symbol time of Gaussian pulse

Specify the product of bandwidth and symbol time for the Gaussian pulse shape. The default is `0.3`. This property requires a real, positive scalar. This property applies when you set the `FrequencyPulse` property to `Gaussian`.

PulseLength

Pulse length

Specify the length of the frequency pulse shape in symbol intervals. The value of this property requires a real, positive integer. The default is `1`.

SymbolPrehistory

Symbol prehistory

Specify the data symbols used by the modulator prior to the first call to the `step` method in reverse chronological order. The default is `1`. This property requires a scalar or vector with odd integer elements between –(`ModulationOrder`–1) and

(ModulationOrder–1). If the value is a vector, then its length must be one less than the value in the PulseLength property.

InitialPhaseOffset

Initial phase offset

Specify the initial phase of the modulated waveform in radians as a real, numeric scalar. The default is 0.

SamplesPerSymbol

Number of samples per output symbol

Specify the upsampling factor at the output as a real, positive, integer scalar. The default is 8. The upsampling factor is the number of output samples that the step method produces for each input sample.

OutputDataType

Data type of output

Specify output data type as one of double | single. The default is double.

Methods

clone	Create CPM modulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset	Reset states of CPM modulator object
step	Modulate using CPM method

Examples

Modulate and demodulate a signal using CPM modulation with Gray mapping and bit inputs.

```
    hMod = comm.CPMModulator(8, 'BitInput', true, ...
                            'SymbolMapping', 'Gray');
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                            'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.CPMDemodulator(8, 'BitOutput', true, ...
                            'SymbolMapping', 'Gray');
% Create an error rate calculator, account for the delay caused by the
    delay = log2(hDemod.ModulationOrder)*hDemod.TracebackDepth;
    hError = comm.ErrorRate('ReceiveDelay', delay);
    for counter = 1:100
        % Transmit 100 3-bit words
        data = randi([0 1],300,1);
        modSignal = step(hMod, data);
        noisySignal = step(hAWGN, modSignal);
        receivedData = step(hDemod, noisySignal);
        errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CPM Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.CPMDemodulator` | `comm.CPFSKModulator` | `comm.MSKModulator` | `comm.GMSKModulator`

comm.CPMModulator.clone

Purpose Create CPM modulator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `CPMModulator` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(*H*)`

comm.CPMModulator.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `CPMModulator` 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.

comm.CPMModulator.release

Purpose Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 CPM modulator object

Syntax

```
reset(H)
```

Description `reset(H)` resets the states of the `CPMModulator` object, `H`.

comm.CPMModulator.step

Purpose Modulate using CPM method

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` modulates input data, `X`, with the CPM modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with data types double, signed integer, or logical. The length of output vector, `Y`, is equal to the number of input samples times the number of samples per symbol specified in the `SamplesPerSymbol` 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.

---

comm.CRCDetector

Purpose

Detect errors in input data using CRC

Description

The `CRCDetector` object computes checksums for its entire input frame.

Construction

`H = comm.CRCDetector` creates a cyclic redundancy code (CRC) detector System object, `H`. This object detects errors in the input data according to a specified generator polynomial.

`H = comm.CRCDetector(Name,Value)` creates a CRC detector 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`).

`H = comm.CRCDetector(POLY,Name,Value)` creates a CRC detector object, `H`. This object has the `Polynomial` property set to `POLY`, and the other specified properties set to the specified values.

Properties

Polynomial

Generator polynomial

Specify the generator polynomial as a binary or integer row vector, with coefficients in descending order of powers. The default is `[1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]`, which is equivalent to vector `[16 12 5 0]`. If you set this property to a binary vector, its length must equal the degree of the polynomial plus `1`. If you set this property to an integer vector, its value must contain the powers of the nonzero terms of the polynomial. For example, `[1 0 0 0 0 0 1 0 1]` and `[8 2 0]` represent the same polynomial, $g(z) = z^8 + z^2 + 1$. The following table lists commonly used generator polynomials.

CRC method	Generator polynomial
CRC-32	`[32 26 23 22 16 12 11 10 8 7 5 4 2 1 0]`
CRC-24	`[24 23 14 12 8 0]`

CRC method	Generator polynomial
CRC-16	`[16 15 2 0]`
Reversed CRC-16	`[16 14 1 0]`
CRC-8	`[8 7 6 4 2 0]`
CRC-4	`[4 3 2 1 0]`

InitialConditions

Initial conditions of shift register

Specify the initial conditions of the shift register as a binary, double or single precision data type scalar or vector. The default is `0`. The vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. When you specify initial conditions as a scalar, the object expands the value to a row vector of length equal to the degree of the generator polynomial.

DirectMethod

Direct method (logical)

When you set this property to `true`, the object uses the direct algorithm for CRC checksum calculations. When you set this property to `false`, the object uses the non-direct algorithm for CRC checksum calculations. The default value for this property is `false`.

Refer to the Communications System Toolbox -> System Design -> Error Detection and Correction -> Cyclic Redundancy Check Coding -> CRC Algorithm section to learn more about the direct and non-direct algorithms.

ReflectInputBytes

Reflect input bytes

Set this property to true to flip the input data on a bytewise basis prior to entering the data into the shift register. When you set this property to `true`, the input frame length divided by the `ChecksumsPerFrame` property value minus the degree of the generator polynomial, which you specify in the `Polynomial` property, must be an integer multiple of `8`. The default value of this property is `false`.

ReflectChecksums

Reflect checksums before final XOR

When you set this property to `true`, the object flips the CRC checksums around their centers after the input data are completely through the shift register. The default value of this property is `false`.

FinalXOR

Final XOR value

Specify the value with which the CRC checksum is to be XORed as a binary scalar or vector. The object applies the XOR operation just prior to appending the input data. The vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. When you specify the final XOR value as a scalar, the object expands the value to a row vector with a length equal to the degree of the generator polynomial. The default value of this property is `0`, which is equivalent to no XOR operation.

ChecksumsPerFrame

Number of checksums per input frame

Specify the number of checksums available at each input frame. The default is `1`. If the length of the input frame to the `step` method equals N and the degree of the generator polynomial equals P, then $N-$`CheckSumsPerFrame`$\times P$ must be divisible by `ChecksumsPerFrame`. The object sets the size of the message word as $N-$`CheckSumsPerFrame`$\times P$, after the checksum bits have been removed from the input frame. This message word corresponds to the first output of the `step` method. The `step` method then outputs a vector, with length equal to the value that you specify in the this property.

For example, you can set the input codeword size to `16` and the generator polynomial to a degree of `3`. Then, you can set the `InitialConditions` property to `0` and the this property to `2` When you do so, the system object:

1. Computes two checksums of size `3`. One checksum comes from the first half of the received codeword, and the other from the second half of the received codeword.

2. Concatenates the two halves of the message word as a single vector of length `10`. Then, outputs this vector through the first output of the step method.

3. Outputs a length `2` binary vector through the second output of the `step` method.

The vector values depend on whether the computed checksums are zero. A `1` in the *i*-th element of the vector indicates that an error occurred in transmitting the corresponding *i*-th segment of the input codeword.

Methods

clone	Create CRC detector object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
reset	Reset states of CRC detector object
step	Detect errors in input data using CRC

Examples

Encode a signal and then detect the errors.

```
% Transmit two message words of length 6
x = logical([1 0 1 1 0 1 0 1 1 1 0 1]');

% Encode the message words using a CRC generator
hGen = comm.CRCGenerator([1 0 0 1], 'ChecksumsPerFrame',2);
codeword = step(hGen, x);

% Add one bit error to each codeword
errorPattern = randerr(2,9,1).';
codewordWithError = xor(codeword, errorPattern(:));

% Decode messages with and without errors using a CRC decoder
hDetect = comm.CRCDetector([1 0 0 1], 'ChecksumsPerFrame',2);
[tx, err] = step(hDetect, codeword);
[tx1, err1] = step(hDetect, codewordWithError);
disp(err)  % err is [0;0], no errors in transmitted message words
disp(err1) % err1 is [1;1], errors in both transmitted message wor
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CRC-N Syndrome Detector block reference page. The object properties correspond to the block parameters.

See Also

`comm.CRCGenerator`

comm.CRCDetector.clone

Purpose

Create CRC detector object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `CRCDetector` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.CRCDetector.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `CRCDetector` 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.

comm.CRCDetector.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 CRC detector object

Syntax

`reset(H)`

Description

`reset(H)` resets the states of the `CRCDetector` object, `H`.

Purpose

Detect errors in input data using CRC

Syntax

`[Y,ERR] = step(H,X)`

Description

`[Y,ERR] = step(H,X)` computes checksums for the entire input frame, `X`. `X` must be a binary column vector and the data type can be double or logical. The step method outputs a row vector `ERR`, with size equal to the number of checksums that you specify in the `CheckSumsPerFrame` property. The elements of `ERR` are `0` if the checksum computation yields a zero value, and `1` otherwise. The method outputs `Y`, with the set of `CheckSumsPerFrame` message words concatenated after removing the checksums bits. The object sets the length of output `Y` as length(`X`) $-P\times$ `CheckSumsPerFrame`, where P is the order of the polynomial that you specify in the `Polynomial` 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 Generate CRC code bits and append to input data

Description The CRCGenerator object generates cyclic redundancy code (CRC) bits for each input data frame and appends them to the frame. The input must be a binary column vector.

Construction H = comm.CRCGenerator creates a cyclic redundancy code (CRC) generator System object, H. This object generates CRC bits according to a specified generator polynomial and appends them to the input data.

H = comm.CRCGenerator(Name,Value) creates a CRC generator 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).

H = comm.CRCGenerator(POLY,Name,Value) creates a CRC generator object, H. This object has the Polynomial property set to POLY, and the other specified properties set to the specified values.

Properties

Polynomial

Generator polynomial

Specify the generator polynomial as a binary or integer row vector, with coefficients in descending order of powers. The default is [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1], which is equivalent to vector [16 12 5 0]. If you set this property to a binary vector, its length must equal the degree of the polynomial plus 1. If you set this property to an integer vector, its value must contain the powers of the nonzero terms of the polynomial. For example, [1 0 0 0 0 0 1 0 1] and [8 2 0] represent the same polynomial, $g(z) = z^8 + z^2 + 1$. The following table lists commonly used generator polynomials.

CRC method	Generator polynomial
CRC-32	`[32 26 23 22 16 12 11 10 8 7 5 4 2 1 0]`
CRC-24	`[24 23 14 12 8 0]`
CRC-16	`[16 15 2 0]`
Reversed CRC-16	`[16 14 1 0]`
CRC-8	`[8 7 6 4 2 0]`
CRC-4	`[4 3 2 1 0]`

InitialConditions

Initial conditions of shift register

Specify the initial conditions of the shift register as a scalar or vector with a binary, double- or single-precision data type. The default is `0`. The vector length must equal the degree of the generator polynomial that you specify in the `Polynomial` property. When you specify initial conditions as a scalar, the object expands the value to a row vector of length equal to the degree of the generator polynomial.

DirectMethod

Direct method (logical)

When you set this property to `true`, the object uses the direct algorithm for CRC checksum calculations. When you set this property to `false`, the object uses the non-direct algorithm for CRC checksum calculations. The default value for this property is `false`.

Refer to the Communications System Toolbox -> System Design -> Error Detection and Correction -> Cyclic Redundancy Check Coding -> CRC Algorithm section to learn more about the direct and non-direct algorithms.

ReflectInputBytes

Reflect input bytes

Set this property to true to flip the input data on a bytewise basis prior to entering the data into the shift register. When you set this property to `true`, the input frame length divided by the `ChecksumsPerFrame` property value must be an integer multiple of `8`. The default value of this property is `false`.

ReflectChecksums

Reflect checksums before final XOR

When you set this property to `true`, the object flips the CRC checksums around their centers after the input data are completely through the shift register. The default value of this property is `false`.

FinalXOR

Final XOR value

Specify the value with which the CRC checksum is to be XORed as a binary scalar or vector. The object applies the XOR operation just prior to appending the input data. The vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. When you specify the final XOR value as a scalar, the object expands the value to a row vector with a length equal to the degree of the generator polynomial. The default value of this property is `0`, which is equivalent to no XOR operation.

ChecksumsPerFrame

Number of checksums per input frame

Specify the number of checksums that the object calculates for each input frame as a positive integer. The default is `1`. The integer must divide the length of each input frame evenly. The object performs the following actions:

1. Divides each input frame into `ChecksumsPerFrame` subframes of equal size.

2. Prefixes the initial conditions vector to each of the subframes.
3. Applies the CRC algorithm to each augmented subframe.
4. Appends the resulting checksums at the end of each subframe.
5. Outputs concatenated subframes.

For example, you can set an input frame size to `10`, the degree of the generator polynomial to `3`, `InitialConditions` property set to `0`, and the `ChecksumsPerFrame` property set to `2`. When you do so, the object divides each input frame into two subframes of size `5` and appends a checksum of size `3` to each subframe. In this example, the output frame has a size $10 + 2 \times 3 = 16$.

Methods

clone	Create CRC generator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of CRC generator object
step	Generate CRC code bits and append to input data

Examples

Encode a signal and then detect the errors.

```
% Transmit two message words of length 6
```

```
x = logical([1 0 1 1 0 1 0 1 1 1 0 1]');

% Encode the message words using a CRC generator
hGen = comm.CRCGenerator([1 0 0 1], 'ChecksumsPerFrame',2);
codeword = step(hGen, x);

% Add one bit error to each codeword
errorPattern = randerr(2,9,1).';
codewordWithError = xor(codeword, errorPattern(:));

% Decode messages with and without errors using a CRC decoder
hDetect = comm.CRCDetector([1 0 0 1], 'ChecksumsPerFrame',2);
[tx, err] = step(hDetect, codeword);
[tx1, err1] = step(hDetect, codewordWithError);
disp(err) % err is [0;0], no errors in transmitted message words
disp(err1) % err1 is [1;1], errors in both transmitted message wor
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CRC-N Generator block reference page. The object properties correspond to the block parameters.

See Also

`comm.CRCDetector`

Purpose Create CRC generator object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `CRCGenerator` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.CRCGenerator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `CRCGenerator` 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.

comm.CRCGenerator.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 CRC generator object

Syntax

`reset(H)`

Description

`reset(H)` resets the states of the `CRCGenerator` object, `H`.

comm.CRCGenerator.step

Purpose Generate CRC code bits and append to input data

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` generates CRC checksums for an input message `X` and appends the checksums to `X`. The input `X` must be a binary column vector and the data type can be double or logical. The length of output `Y` is length(`X`) + $P\times$ `CheckSumsPerFrame`, where P is the order of the polynomial that you specify in the `Polynomial` 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

Demodulate using DBPSK method

Description

The `DBPSKDemodulator` object demodulates a signal that was modulated using the differential binary phase shift keying method. The input is a baseband representation of the modulated signal.

Construction

`H = comm.DBPSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the differential binary phase shift keying (DBPSK) method.

`H = comm.DBPSKDemodulator(Name,Value)` creates a DBPSK demodulator 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`).

`H = comm.DBPSKDemodulator(PHASE,Name,Value)` creates a DBPSK demodulator object, `H`. This object has the `PhaseRotation` property set to `PHASE` and the other specified properties set to the specified values.

Properties

PhaseRotation

Additional phase shift

Specify the additional phase difference between previous and current modulated bits in radians as a real scalar. The default is `0`. This value corresponds to the phase difference between previous and current modulated bits when the input is zero.

OutputDataType

Data type of output

Specify output data type as one of `Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | logical`. The default is `Full precision`. When you set this property to `Full precision`, the output data type has the same data type as the input. In this case, that value must be a double- or single-precision data type.

Methods

clone	Create DBPSK demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of DBPSK demodulator object
step	Demodulate using DBPSK method

Examples

Modulate and demodulate a signal using DBPSK modulation.

```
    hMod = comm.DBPSKModulator(pi/4);
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                         'Signal to noise ratio (SNR)','SNR',15);
    hDemod = comm.DBPSKDemodulator(pi/4);
% Create an error rate calculator, account for the one bit transient caus
    hError = comm.ErrorRate('ComputationDelay',1);
    for counter = 1:100
        % Transmit a 50-symbol frame
        data = randi([0 1],50,1);
        modSignal = step(hMod, data);
        noisySignal = step(hAWGN, modSignal);
        receivedData = step(hDemod, noisySignal);
        errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the DBPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.DBPSKModulator` | `comm.DQPSKModulator`

comm.DBPSKDemodulator.clone

Purpose

Create DBPSK demodulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `DBPSKDemodulator` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.DBPSKDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`dTF = isLocked(H)` returns the locked status, `TF` of the `DBPSKDemodulator` 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.

comm.DBPSKDemodulator.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 DBPSK demodulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `DBPSKDemodulator` object, `H`.

comm.DBPSKDemodulator.step

Purpose Demodulate using DBPSK method

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` demodulates input data, `X`, with the DBPSK demodulator System object, `H`, and returns `Y`. Input `X` must be a double or single precision data type scalar or column vector.

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

Modulate using DBPSK method

Description

The `DBPSKModulator` object modulates using the differential binary phase shift keying method. The output is a baseband representation of the modulated signal.

Construction

`H = comm.DBPSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the differential binary phase shift keying (DBPSK) method.

`H = comm.DBPSKModulator(Name,Value)` creates a DBPSK modulator 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`).

`H = comm.DBPSKModulator(PHASE,Name,Value)` creates a DBPSK modulator object, `H`. This object has the `PhaseRotation` property set to `PHASE`, and the other specified properties set to the specified values.

Properties

PhaseRotation

Additional phase shift

Specify the additional phase difference between previous and current modulated bits in radians as a real scalar value. The default is `0`. This value corresponds to the phase difference between previous and current modulated bits when the input is zero.

OutputDataType

Data type of output

Specify output data type as one of `double | single`. The default is `double`.

Methods

clone	Create DBPSK modulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of DBPSK modulator object
step	Modulate using DBPSK method

Examples

Modulate data using DBPSK modulation, and visualize the data in a scatter plot.

```
% Create binary data symbols
    data = randi([0 1], 96, 1);
% Create a DBPSK modulator System object and set the phase rotation to pi
    hModulator = comm.DBPSKModulator(pi/4);
% Modulate and plot the data
    modData = step(hModulator, data);
    scatterplot(modData)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the DBPSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

comm.DBPSKDemodulator | comm.DQPSKModulator

Purpose

Create DBPSK modulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `DBPSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.DBPSKModulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.DBPSKModulator.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `DBPSKModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.DBPSKModulator.reset

Purpose Reset states of DBPSK modulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `DBPSKModulator` object, `H`.

Purpose

Modulate using DBPSK method

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` modulates input data, `X`, with the DBPSK modulator System object, `H`. It returns the baseband modulated output, `Y`. The input must be a numeric or logical data type column vector of bits.

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.

---

comm.Descrambler

Purpose

Descramble input signal

Description

The `Descrambler` object descrambles a scalar or column vector input signal. The Descrambler object is the inverse of the Scrambler object. If you use the Scrambler object in a transmitter, then you use the Descrambler object in the related receiver.

Construction

`H = comm.Descrambler` creates a descrambler System object, `H`. This object descrambles the input data using a linear feedback shift register that you specify with the Polynomial property.

`H = comm.Descrambler(Name,Value)` creates a descrambler 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`).

`H = comm.Descrambler(N,POLY,COND,Name,Value)` creates a descrambler object, `H`. This object has the `CalculationBase` property set to `N`, the `Polynomial` property set to `POLY`, the `InitialConditions` property set to `COND`, and the other specified properties set to the specified values.

Properties

CalculationBase

Range of input data

Specify calculation base as a positive, integer, scalar value. The step method input and output integers are in the range [0, CalculationBase–1]. The default is `4`.

Polynomial

Linear feedback shift register connections

Specify the polynomial that determines the shift register feedback connections. The default is `[1 1 1 0 1]`. You can the generator polynomial as a numeric, binary vector that lists the coefficients of the polynomial in order of ascending powers of z^{-1}, where $p(z^{-1}) = 1 + p1z^{-1} + p2z^{-2} + ...$ is the generator polynomial. The first and last elements must be `1`. Alternatively, you can specify the generator polynomial as a numeric vector. This vector contains the exponents of z^{-1} for the nonzero terms of the polynomial, in order of ascending powers of z^{-1}. In this case, the first vector element must be `0`. For example, both `[1 0 0 0 0 0 1 0 1]` and `[0 -6 -8]` specify the same polynomial $p(z^{-1}) = 1 + z^{-6} + z^{-8}$.

InitialConditions

Initial values of linear feedback shift register

Specify the initial values of the linear feedback shift register as an integer row vector with values in [0 CalculationBase–1]. The default is `[0 1 2 3]`. The length of this property vector must equal the order of the `Polynomial` property vector.

Methods

clone	Create descrambler object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of descrambler object
step	Descramble input signal

Examples

Scramble and descramble random data with values in the range [0 7].

```
% Create scrambler and descrambler objects with calculation base o
N = 8;
hSCR = comm.Scrambler(N, [1 0 1 1 0 1 0 1],...
```

```
        [0 3 2 2 5 1 7]);
hDSCR = comm.Descrambler(N, [1 0 1 1 0 1 0 1],...
        [0 3 2 2 5 1 7]);
for counter = 1:10
  data = randi([0 N-1], 4, 1);
  scrData = step(hSCR, data);
  deScrData = step(hDSCR, scrData);
  [data, scrData, deScrData]
end
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Descrambler block reference page. The object properties correspond to the block parameters.

See Also

`comm.Scrambler` | `comm.PNSequence`

Purpose

Create descrambler object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `Descrambler` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.Descrambler.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.Descrambler.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `Descrambler` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.Descrambler.reset

Purpose Reset states of descrambler object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `Descrambler` object, `H`.

Purpose

Descramble input signal

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` descrambles input data, `X`, and returns the result in `Y`. `X` must be a double precision, logical, or integer column vector. The output `Y` is same data type and length as the input vector, `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

Decode binary signal using differential decoding

Description

The `DifferentialDecoder` object decodes the binary input signal. The output is the logical difference between the consecutive input element within a channel.

Construction

`H = comm.DifferentialDecoder` creates a differential decoder System object, `H`. This object decodes a binary input signal that was previously encoded using a differential encoder.

`H = comm.DifferentialDecoder(Name,Value)` creates 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

InitialCondition

Initial value used to generate initial output

Specify the initial condition as a real scalar. This property can have a logical, numeric, or fixed-point (embedded.fi object) data type. The default is `0`. The object treats nonbinary values as binary signals.

Methods

clone	Create differential decoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset	Reset states of differential decoder object
step	Decode binary signal using differential decoding

Examples

Decode a differentially encoded signal.

```
% Create Differential Encoder System object
    hdiffenc = comm.DifferentialEncoder;
% Create Differential Decoder System object
    hdiffdec = comm.DifferentialDecoder;
% Generate random binary data
    data = randi([0 1], 100, 1);
% Encode data
    encdata = step(hdiffenc,data);
% Decode data
    recdata = step(hdiffdec, encdata);
    errors = biterr(data, recdata);
    fprintf(1, ['\nThere were %d errors in the decoded signal ' ...
                'out of %d bits\n'],errors, length(data));
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Differential Decoder block reference page. The object properties correspond to the block parameters, except:

The object only supports single channel, column vector inputs.

See Also

comm.DifferentialEncoder

comm.DifferentialDecoder.clone

Purpose

Create differential decoder object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `DifferentialDecoder` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.DifferentialDecoder.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `DifferentialDecoder` 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.

comm.DifferentialDecoder.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 differential decoder object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `DifferentialDecoder` object, `H`.

comm.DifferentialDecoder.step

Purpose

Decode binary signal using differential decoding

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` decodes the differentially encoded input data, `X`, and returns the decoded data, `Y`. The input `X` must be a column vector of data type logical, numeric, or fixed-point (embedded.fi objects). `Y` has the same data type as `X`. The object treats non-binary inputs as binary signals. The object computes the initial output value by performing an Xor operation of the value in the `InitialCondition` property and the first element of the vector you input the first time you call the 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

Encode binary signal using differential coding

Description

The `DifferentialEncoder` object encodes the binary input signal within a channel. The output is the logical difference between the current input element and the previous output element.

Construction

`H = comm.DifferentialEncoder` creates a differential encoder System object, `H`. This object encodes a binary input signal by calculating its logical difference with the previously encoded data.

`H = comm.DifferentialEncoder(Name,Value)` creates 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

InitialCondition

Initial value used to generate initial output

Specify the initial condition as a real scalar. This property can have a logical, numeric, or fixed-point (embedded.fi object) data type. The default is `0`. The object treats nonbinary values as binary signals.

Methods

clone	Create differential encoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset	Reset states of differential encoder object
step	Encode binary signal using differential coding

Examples

Encode binary signal using differential coding.

```
% Create Differential Encoder System object
    hdiffenc = comm.DifferentialEncoder;
% Generate random binary data
    data = randi([0 1], 11, 1);
% Encode data
    encdata = step(hdiffenc,data);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Differential Encoder block reference page. The object properties correspond to the block parameters, except:

The object only supports single channel, column vector inputs.

See Also

`comm.DifferentialDecoder`

Purpose

Create differential encoder object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `DifferentialEncoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.DifferentialEncoder.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.DifferentialEncoder.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `DifferentialEncoder` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.DifferentialEncoder.reset

Purpose Reset states of differential encoder object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `DifferentialEncoder` object, `H`.

Purpose

Encode binary signal using differential coding

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` encodes the binary input data, `X`, and returns the differentially encoded data, `Y`. The input `X` must be a column vector of data type logical, numeric, or fixed-point (embedded.fi objects). `Y` has the same data type as `X`. The object treats non-binary inputs as binary signals. The object computes the initial output value by performing an Xor operation of the value in the `InitialCondition` property and the first element of the vector you input the first time you call the 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

Generate variable frequency sinusoid

Description

The `DiscreteTimeVCO` (voltage-controlled oscillator) object generates a signal whose frequency shift from the quiescent frequency property is proportional to the input signal. The input signal is interpreted as a voltage.

Construction

`H = comm.DiscreteTimeVCO` creates a discrete-time voltage-controlled oscillator (VCO) System object, `H`. This object generates a sinusoidal signal with the frequency shifted from the specified quiescent frequency to a value proportional to the input signal.

`H = comm.DiscreteTimeVCO(Name,Value)` creates a discrete-time VCO 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

OutputAmplitude

Amplitude of output signal

Specify the amplitude of the output signal as a double- or single-precision, scalar value. The default is `1`. This property is tunable.

QuiescentFrequency

Frequency of output signal when input is zero

Specify the quiescent frequency of the output signal in Hertz, as a double- or single-precision, real, scalar value. The default is `10`. This property is tunable.

Sensitivity

Sensitivity of frequency shift of output signal

Specify the sensitivity of the output signal frequency shift to the input as a double- or single-precision, real, scalar value. The default is `1`. This value scales the input voltage and, consequently, the shift from the quiescent frequency value. The property measures Sensitivity in Hertz per volt. This property is tunable.

InitialPhase

Initial phase of output signal

Specify the initial phase of the output signal, in radians, as a double or single precision, real, scalar value. The default is `0`.

SampleRate

Sample rate of input

Specify the sample rate of the input, in Hertz, as a double- or single-precision, positive, scalar value. The default is `100`.

Methods

clone	Create discrete-time VCO object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of discrete-time VCO object
step	Generate variable frequency sinusoid

Examples

Generate an FSK signal using a discrete time VCO.

```
% Create a SignalSource System object and generate random data
hreader = dsp.SignalSource;
```

```
hreader.Signal = randi([0 7],10,1);
% Rectangular pulse shaping
hreader.Signal = rectpulse(hreader.Signal, 100);
% Create a signal logger System object
hlogger = dsp.SignalSink;
% Create a discrete time VCO object and generate an FSK signal
hdvco = comm.DiscreteTimeVCO('OutputAmplitude',8, ...
                             'QuiescentFrequency',1);
while(~isDone(hreader))
  sig = step(hreader);
  y = step(hdvco,sig);
  step(hlogger,y);
end
oscsig = hlogger.Buffer;
% Plot FSK signal
t = [0:length(oscsig)-1]'/hdvco.SampleRate;
plot(t,hreader.Signal,'--r', 'LineWidth',3); hold on;
plot(t,oscsig,'-b'); hold off;
xlabel('time (s)');
legend('Input Signal', 'FSK Signal');
```

Algorithms

This object implements the algorithm, inputs, and outputs as described on the Discrete-Time VCO block reference page. However, this object and the corresponding block may not generate the exact same outputs for single-precision inputs or property values due to the following differences in casting strategies and arithmetic precision issues:

- The block always casts the result of intermediate mathematical operations to the input data type. The object does not cast intermediate results and MATLAB decides the data type. The object casts the final output to the input data type.
- You can specify the `SampleRate` object property in single-precision or double-precision. The block does not allow this.
- In arithmetic operations with more than two operands with mixed data types, the result may differ depending on the order of operation.

Thus, the following calculation may also contribute to the difference in the output of the block and the object:

input * sensitivity * `sampleTime`

- The block performs this calculation from left to right. However, since sensitivity * `sampleTime` is a one-time calculation, the object calculates this in the following manner:

  input * (sensitivity * `sampleTime`)

See Also

`comm.CPMCarrierPhaseSynchronizer` | `comm.PSKCarrierPhaseSynchronizer`

Purpose Create discrete-time VCO object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `DiscreteTimeVCO` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.DiscreteTimeVCO.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `DiscreteTimeVCO` 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.

comm.DiscreteTimeVCO.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 discrete-time VCO object

Syntax

```
reset(H)
```

Description `reset(H)` resets the states of the `DiscreteTimeVCO` object, `H`.

comm.DiscreteTimeVCO.step

Purpose

Generate variable frequency sinusoid

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` generates a sinusoidal signal, `Y`, with frequency shifted, from the value you specify in the `QuiescentFrequency` property, to a value proportional to the input signal, `X`. The input, `X`, must be a double or single precision, real, scalar value. The output, `Y`, has the same data type and size as 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

Demodulate using M-ary DPSK method

Description

The `DPSKDemodulator` object demodulates a signal that was modulated using the M-ary differential phase shift keying method. The input is a baseband representation of the modulated signal. The input and output for this object are discrete-time signals. This object accepts a scalar-valued or column vector input signal.

Construction

`H = comm.DPSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the *M*-ary differential phase shift keying (M-DPSK) method.

`H = comm.DPSKDemodulator(Name,Value)` creates an M-DPSK demodulator 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`).

`H = comm.DPSKDemodulator(M,PHASE,Name,Value)` creates an M-DPSK demodulator object, `H`. This object has the `ModulationOrder` property set to `M`, the `PhaseRotation` property set to `PHASE`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is `8`.

PhaseRotation

Additional phase shift

Specify the additional phase difference between previous and current modulated symbols in radians as a real scalar value. The default is `pi/8`. This value corresponds to the phase difference between previous and current modulated symbols when the input is zero.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. The default is `false`. When you set this property to `true` the `step` method outputs a column vector of bit values. The length of this column vector is equal to **log2**(`ModulationOrder`) times the number of demodulated symbols.

When you set this property to `false`, the `step` method outputs a column vector. The length of this column vector is equal to that of the input data vector. The output contains integer symbol values between `0` and `ModulationOrder-1`.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder`) bits to the corresponding symbol as one of `Binary | Gray`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the input integer m, between ($0 \leq m \leq$ `ModulationOrder`–1) maps to the current symbol. This mapping uses **exp**($j\times$ `PhaseRotation` + $j\times 2\times\pi\times$m/ModulationOrder)$\times$(previously modulated symbol).

OutputDataType

Data type of output

Specify the output data type as one of `Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32`. The default is `Full precision`. When you set this property to `Full precision`, the input data type is single or double precision, the output data is the same as that of the input. When you set the `BitOutput` property to true, `logical` data type becomes a valid option.

Methods

clone	Create M-DPSK demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of M-DPSK demodulator object
step	Demodulate using M-ary DPSK method

Examples

Modulate and demodulate a signal using 8-DPSK modulation.

```
    hMod = comm.DPSKModulator(8,pi/4);
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                             'Signal to noise ratio (SNR)','SNR',20);
    hDemod = comm.DPSKDemodulator(8,pi/4);
% Create an error rate calculator, account for the one symbol transien
    hError = comm.ErrorRate('ComputationDelay',1);
    for counter = 1:100
% Transmit a 50-symbol frame
        data = randi([0 hMod.ModulationOrder-1],50,1);
        modSignal = step(hMod, data);
        noisySignal = step(hAWGN, modSignal);
        receivedData = step(hDemod, noisySignal);
        errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-DPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.DPSKModulator` | `comm.DBPSKDemodulator` | `comm.DQPSKDemodulator`

Purpose Create M-DPSK demodulator object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `DPSKDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.DPSKDemodulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.DPSKDemodulator.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `DPSKDemodulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.DPSKDemodulator.reset

Purpose

Reset states of M-DPSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DPSKDemodulator` object, `H`.

Purpose

Demodulate using M-ary DPSK method

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` demodulates input data, `X`, with the DPSK demodulator System object, `H`, and returns `Y`. Input `X` must be a double or single precision data type scalar or column vector. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

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

Modulate using M-ary DPSK method

Description

The `DPSKModulator` object modulates using the M-ary differential phase shift keying method. The output is a baseband representation of the modulated signal.

Construction

`H = comm.DPSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the M-ary differential phase shift keying (M-DPSK) method.

`H = comm.DPSKModulator(Name,Value)` creates an M-DPSK modulator 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`).

`H = comm.DPSKModulator(M,PHASE,Name,Value)` creates an M-DPSK modulator object, `H`. This object has the `ModulationOrder` property set to `M`, the `PhaseRotation` property set to `PHASE`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is `8`.

PhaseRotation

Additional phase shift

Specify the additional phase difference between previous and current modulated symbols in radians as a real scalar value. The default is `pi/8`. This value corresponds to the phase difference between previous and current modulated symbols when the input is zero.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`. When you set this property to `true`, the `step` method input must be a column vector of bit values whose length is an integer multiple of **log2**(`ModulationOrder`). This vector contains bit representations of integers between `0` and `ModulationOrder–1`. When you set this property to `false`, the `step` method input requires a column vector of integer symbol values between `0` and `ModulationOrder–1`.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder`) input bits to the corresponding symbol as one of `Binary | Gray`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the input integer m, between ($0 \le m \le$ `ModulationOrder–1`) shifts the output phase. This shift is (`PhaseRotation` + $2 \times \pi \times m$/`ModulationOrder`) radians from the previous output phase. The output symbol uses **exp**($j\times$ `PhaseRotation` + $j\times 2\times\pi\times$ m/ModulationOrder)$\times$(previously modulated symbol).

OutputDataType

Data type of output

Specify output data type as one of `double | single`. The default is `double`.

Methods

clone	Create M-DPSK modulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of M-DPSK modulator object
step	Modulate using M-ary DPSK method

Examples

Modulate data using 8-DPSK modulation and visualize the data in a scatter plot.

```
% Create binary data for 1000, 3 bit symbols
    data = randi([0 1],3000,1);
% Create an 8-DPSK modulator System object with bits as inputs,phase rota
    hModulator = comm.DPSKModulator(8,pi/4,'BitInput',true);
% Modulate and plot the data
    modData = step(hModulator, data);
    scatterplot(modData)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-DPSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.DPSKDemodulator` | `comm.DBPSKModulator` | `comm.DQPSKModulator`

Purpose

Create M-DPSK modulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `DPSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.DPSKModulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.DPSKModulator.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `DPSKModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.DPSKModulator.reset

Purpose Reset states of M-DPSK modulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `DPSKModulator` object, `H`.

comm.DPSKModulator.step

Purpose

Modulate using M-ary DPSK method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` modulates input data, `X`, with the DPSK modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with numeric or logical data types.

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

Demodulate using DQPSK method

Description

The `DQPSKDemodulator` object demodulates a signal that was modulated using the differential quaternary phase shift keying method. The input is a baseband representation of the modulated signal.

Construction

`H = comm.DQPSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the differential quadrature phase shift keying (DQPSK) method.

`H = comm.DQPSKDemodulator(Name,Value)` creates a DQPSK demodulator 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`).

`H = comm.DQPSKDemodulator(PHASE,Name,Value)` creates a DQPSK demodulator object, `H`. This object has the `PhaseRotation` property set to `PHASE` and the other specified properties set to the specified values.

Properties

PhaseRotation

Additional phase shift

Specify the additional phase difference between previous and current modulated symbols in radians as a real scalar. The default is `pi/4`. This value corresponds to the phase difference between previous and current modulated symbols when the input is zero.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. The default is `false`. When you set this property to `true` the `step` method outputs a column vector of bit values with length equal to twice the number of demodulated symbols. When you set this property to `false`, the `step` method outputs a column vector, of length equal to the input data vector, that contains integer symbol values between `0` and `3`.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of 2 bits to the corresponding symbol as one of `Binary | Gray`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the integer m, between $0 \le m \le 3$ maps to the current symbol as $\mathbf{exp}(j \times \texttt{PhaseRotation} + j \times 2 \times \pi \times {}^{m}\!/_{4}) \times$ (previously modulated symbol).

OutputDataType

Data type of output

Specify the output data type as one of `Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32`. The default is `Full precision`. When you set this property to `Full precision` the output has the same data type as that of the input. In this case, the input data type is single- or double-precision value. When you set the `BitOutput` property to `true`, `logical` data type becomes a valid option.

Methods

clone	Create DQPSK demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset	Reset states of DQPSK demodulator object
step	Demodulate using DQPSK method

Examples

Modulate and demodulate a signal using DQPSK modulation.

```
    hMod = comm.DQPSKModulator(pi/8);
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                          'Signal to noise ratio (SNR)','SNR',15);
    hDemod = comm.DQPSKDemodulator(pi/8);
% Create an error rate calculator, account for the one symbol transient c
    hError = comm.ErrorRate('ComputationDelay',1);
    for counter = 1:100
% Transmit a 50-symbol frame
        data = randi([0 3],50,1);
        modSignal = step(hMod, data);
        noisySignal = step(hAWGN, modSignal);
        receivedData = step(hDemod, noisySignal);
        errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the DQPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

comm.DQPSKModulator | comm.DPSKDemodulator | comm.DBPSKDemodulator

Purpose

Create DQPSK demodulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `DQPSKDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.DQPSKDemodulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.DQPSKDemodulator.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `DQPSKDemodulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.DQPSKDemodulator.reset

Purpose Reset states of DQPSK demodulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `DQPSKDemodulator` object, `H`.

Purpose Demodulate using DQPSK method

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` demodulates input data, `X`, with the DQPSK demodulator System object, `H`, and returns `Y`. Input `X` must be a single or double precision data type scalar or column vector. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

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

Modulate using DQPSK method

Description

The `DQPSKModulator` object modulates using the differential quaternary phase shift keying method. The output is a baseband representation of the modulated signal.

Construction

`H = comm.DQPSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the differential quadrature phase shift keying (DQPSK) method.

`H = comm.DQPSKModulator(Name,Value)` creates a DQPSK modulator 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`).

`H = comm.DQPSKModulator(PHASE,Name,Value)` creates a DQPSK modulator object, `H`. This object has the `PhaseRotation` property set to `PHASE` and the other specified properties set to the specified values.

Properties

PhaseRotation

Additional phase shift

Specify the additional phase difference between previous and current modulated symbols in radians as a real scalar value. The default is `pi/4`. This value corresponds to the phase difference between previous and current modulated symbols when the input is zero.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`. When you set this property to true, the step method input must be a column vector of bit values. The length of this vector is an integer multiple of two. This vector contains bit representations of integers between `0` and `3`. When you set this property to `false`, the `step` method input must be a column vector of integer symbol values between `0` and `3`.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of two input bits to the corresponding symbol as one of `Binary | Gray`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the input integer m, between $0 \leq m \leq 3$ shifts the output phase. This shift is (`PhaseRotation` + $2 \times \pi \times m/4$) radians from the previous output phase. The output symbol is $\mathbf{exp}(j \times \texttt{PhaseRotation} + j \times 2 \times \pi \times m/4) \times$ (previously modulated symbol).

OutputDataType

Data type of output

Specify output data type as one of `double | single`. The default is `double`.

Methods

clone	Create DQPSK modulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset	Reset states of DQPSK modulator object
step	Modulate using DQPSK method

Examples

Modulate data using DQPSK modulation and visualize the data in a scatter plot.

```
% Create binary data for 100, 4 bit symbols
   data = randi([0 1],400,1);
% Create a DQPSK modulator System object with bits as inputs,phase rotat
   hModulator = comm.DQPSKModulator(pi/8,'BitInput',true);
% Modulate and plot the data
   modData = step(hModulator, data);
   scatterplot(modData)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the DQPSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

comm.DQPSKDemodulator | comm.DPSKModulator | comm.DBPSKModulator

Purpose

Create DQPSK modulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `DQPSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.DQPSKModulator.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.DQPSKModulator.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `DQPSKModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.DQPSKModulator.reset

Purpose Reset states of DQPSK modulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `DQPSKModulator` object, `H`.

Purpose

Modulate using DQPSK method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` modulates input data, `X`, with the DQPSK modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with numeric or logical data types.

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 Recover symbol timing phase using early-late gate method

Description The `EarlyLateGateTimingSynchronizer` object recovers the symbol timing phase of the input signal using the early-late gate method. This object implements a non-data-aided feedback method.

Construction `H = comm.EarlyLateGateTimingSynchronizer` creates a timing phase synchronizer System object, `H`. This object recovers the symbol timing phase of the input signal using the early-late gate method.

`H = comm.EarlyLateGateTimingSynchronizer(Name,Value)` creates an early-late gate timing synchronizer 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

SamplesPerSymbol

Number of samples representing each symbol

Specify the number of samples that represent each symbol in the input signal as an integer-valued scalar greater than `1`. The default is `4`.

ErrorUpdateGain

Error update step size

Specify the step size for updating successive timing phase estimates as a positive real scalar value. Typically, this number is less than 1/`SamplesPerSymbol`, which corresponds to a slowly varying timing phase. The default is `0.05`. This property is tunable.

ResetInputPort

Enable synchronization reset input

Set this property to `true` to enable resetting the timing phase recovery process based on an input argument value. When you set this property to `true`, you must specify a reset input value to the `step` method. When the reset input is a nonzero value, the object restarts the timing phase recovery process. When you set this property to `false`, the object does not restart. The default is `false`.

ResetCondition

Condition for timing phase recovery reset

Specify the conditions to reset the timing phase recovery process as one of `Never | Every frame`. The default is `Never`. When you set this property to `Never`, the phase recovery process never restarts. The object operates continuously, retaining information from one symbol to the next. When you set this property to `Every frame`, the timing phase recovery restarts at the start of each frame of data. In this case, each time the object calls the `step` method. This property applies when you set the `ResetInputPort` property to `false`.

Methods

clone	Create early-late gate timing phase synchronizer object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of early-late gate timing phase synchronizer
step	Recover symbol timing phase using early-late gate method

Examples

Recover timing phase using the early-late gate method.

```
 % Initialize data
   L = 16; M = 16; numSymb = 100; snrdB = 30;
   R = 25; rollOff = 0.75; filtDelay = 3; g = 0.07; delay = 6.6498;

 % Design raised cosine filters
   txFiltSpec = fdesign.pulseshaping(L, 'Square root raised cosine', ...
                       'Nsym,Beta', 2*filtDelay, rollOff);
   txFilterDesign = design(txFiltSpec);
   txFilterDesign.Numerator = sqrt(L)*txFilterDesign.Numerator;
 % Create System objects
   hMod = comm.RectangularQAMModulator(M, ...
                   'NormalizationMethod', 'Average power');
   hTxFilter = dsp.FIRInterpolator(L, txFilterDesign.Numerator);
   hDelay = dsp.VariableFractionalDelay('MaximumDelay', L);
   hChan = comm.AWGNChannel('NoiseMethod',  ...
                   'Signal to noise ratio (SNR)', 'SNR', snrdB, ...
                   'SignalPower', 1/L);
   hRxFilter = dsp.DigitalFilter(...
                   'TransferFunction', 'FIR (all zeros)', ...
                   'Numerator', txFilterDesign.Numerator);
   hSync = comm.EarlyLateGateTimingSynchronizer(...
                       'SamplesPerSymbol', L, ...
                       'ErrorUpdateGain', g);
 % Generate random data
   data = randi([0 M-1], numSymb, 1);

% Modulate and filter transmitter data.
   modData = step(hMod, data);
   filterData = step(hTxFilter, modData);

% Introduce a random delay and add noise
   delayedData = step(hDelay, filterData, delay);
   chData = step(hChan, delayedData);

% Filter receiver data.
```

```
   rxData = step(hRxFilter, chData);

% Estimate the delay from the received signal
   [~, phase] = step(hSync, rxData);
fprintf(1, 'Actual Timing Delay: %f\n', delay);
fprintf(1, 'Estimated Timing Delay: %f\n', phase(end));
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Early-Late Gate Timing Recovery block reference page. The object properties correspond to the block parameters, except:

The block **Reset** parameter corresponds to the `ResetInputPort` and `ResetCondition` properties.

See Also

`comm.GardnerTimingSynchronizer` | `comm.MSKTimingSynchronizer`

comm.EarlyLateGateTimingSynchronizer.clone

Purpose Create early-late gate timing phase synchronizer object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `EarlyLateGateTimingSynchronizer` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.EarlyLateGateTimingSynchronizer.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` method returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`.

comm.EarlyLateGateTimingSynchronizer.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` method returns a positive integer, `N`, representing the number of outputs from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `EarlyLateGateTimingSynchronizer` 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.

comm.EarlyLateGateTimingSynchronizer.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 early-late gate timing phase synchronizer

Syntax `reset(H)`

Description `reset(H)` resets the states of early-late gate timing phase synchronizer for the `EarlyLateGateTimingSynchronizer` object `H`.

comm.EarlyLateGateTimingSynchronizer.step

Purpose

Recover symbol timing phase using early-late gate method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Description

`[Y,PHASE] = step(H,X)` performs timing phase recovery and returns the time-synchronized signal, `Y`, and the estimated timing phase, `PHASE`, for input signal `X`. The input `X` must be a double or single precision complex column vector. Ideally, it is when the timing phase estimate is zero and the input signal has symmetric Nyquist pulses. In this case, the timing error detector for the early-late gate method requires samples that span one symbol interval.

`[Y,PHASE] = step(H,X,R)` restarts the timing phase recovery process when you input a reset signal, `R`, that is non-zero. `R` must be a double precision or logical scalar. This syntax applies 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.

Purpose

Compute bit or symbol error rate of input data

Description

The `ErrorRate` object compares input data from a transmitter with input data from a receiver and calculates the error rate as a running statistic. To obtain the error rate, the object divides the total number of unequal pairs of data elements by the total number of input data elements from one source.

Construction

`H = comm.ErrorRate` creates an error rate calculator System object, `H`. This object computes the error rate of the received data by comparing it to the transmitted data.

`H = comm.ErrorRate(Name,Value)` creates an error rate calculator 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

ReceiveDelay

Number of samples to delay transmitted signal

Specify the number of samples by which the received data lags behind the transmitted data. This value must be a real, nonnegative, double-precision, integer scalar. Use this property to align the samples for comparison in the transmitted and received input data vectors. Specify the delay in number of samples, regardless of whether the input is a scalar or a vector. The default is `0`.

ComputationDelay

Computation delay

Specify the number of data samples that the object should ignore at the beginning of the comparison. This value must be a real, nonnegative, double-precision, integer scalar. Use this property to ignore the transient behavior of both input signals. The default is `0`.

Samples

Samples to consider

Specify samples to consider as one of `Entire frame` | `Custom` | `Input port`. The property defines whether the object should consider all or only part of the input frames when computing error statistics. The default is `Entire frame`. Select `Entire frame` to compare all the samples of the RX frame to those of the TX frame. Select `Custom` or `Input port` to list the indices of the RX frame elements that the object should consider when making comparisons. When you set this property to `Custom`, you can list the indices as a scalar or a column vector of double-precision integers through the `CustomSamples` property. When you set this property to `Input port`, you can list the indices as an input to the `step` method.

CustomSamples

Selected samples from frame

Specify a scalar or a column vector of double-precision, real, positive integers. This value lists the indices of the elements of the RX frame vector that the object uses when making comparisons. This property applies when you set the `Samples` property to `Custom`. The default is an empty vector, which specifies that all samples are used.

ResetInputPort

Enable error rate reset input

Set this property to `true` to reset the error statistics via an input to the `step` method. The default is `false`.

Methods

clone	Create error rate calculator object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of error rate calculator object
step	Compute bit or symbol error rate of input data

Examples

Calculate BER between transmitted and received signal.

```
% Use 8-DPSK modulation in an AWGN channel
    hMod = comm.DPSKModulator('ModulationOrder',8,'BitInput',true);
    hDemod = comm.DPSKDemodulator('ModulationOrder',8,'BitOutput',true
    hAWGN = comm.AWGNChannel('NoiseMethod',...
                         'Signal to noise ratio (SNR)','SNR', 7);

% Create an error rate calculator, accounting for the three bit
% (i.e., one symbol) transient caused by the differential modulation
    hError = comm.ErrorRate('ComputationDelay',3);
    BER = zeros(10,1);

% Calculate BER for 10 frames
    for i= 1:10
        data = randi([0 1], 96, 1);
        modData = step(hMod, data);
        receivedSignal = step(hAWGN, modData);
        receivedData = step(hDemod, receivedSignal);
        errors = step(hError, data, receivedData);
        BER(i) = errors(1);
    end
    disp(BER) % display BER for 10 frames
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Error Rate Calculation block reference page. The object properties correspond to the block parameters, except:

- The **Output data** and **Variable name** block parameters do not have a corresponding properties. The object always returns the result as an output.
- The **Stop simulation** block parameter does not have a corresponding property. To implement similar behavior, use the output of the `step` method in a while loop, to programmatically stop the simulation. See the Gray Coded 8-PSK.
- The **Computation mode** parameter corresponds to the `Samples` and `CustomSamples` properties.

See Also

`alignsignals` | `finddelay`

Purpose

Create error rate calculator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `ErrorRate` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.ErrorRate.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.ErrorRate.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `ErrorRate` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.ErrorRate.reset

Purpose Reset states of error rate calculator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `ErrorRate` object, `H`.

Purpose Compute bit or symbol error rate of input data

Syntax

```
Y = step(H,TX,RX)
Y = step(H,TX,RX,SEL)
Y = step(H,TX,RX,RST)
```

Description

`Y = step(H,TX,RX)` counts the number of differences between the transmitted data vector, `TX`, and received data vector, `RX`. The step method outputs a three-element vector consisting of the error rate, followed by the number of errors detected and the total number of samples compared. `TX` and `RX` inputs can be either scalars or column vectors of the same data type. Valid data types are single, double, integer or logical. If `TX` is a scalar and `RX` is a vector, or vice-versa, then the block compares the scalar with each element of the vector.

`Y = step(H,TX,RX,SEL)` calculates the errors based on selected samples from the input frame specified by the `SEL` input. `SEL` must be a real, double-precision integer-valued scalar or a column vector. The vector lists the indices of the elements of the `RX` input vector that the object should consider when making comparisons. This syntax applies when you set the `Samples` property to 'Input Port'.

`Y = step(H,TX,RX,RST)` resets the error count whenever the input `RST` is non-zero. `RST` must be a real, double, or logical scalar. When you set the `RST` input to a nonzero value, the object clears its error statistics and then recomputes them based on the current `TX` and `RX` inputs. This syntax applies when you set the `ResetInputPort` property to true. You can combine optional input arguments when their enabling properties are set. Optional inputs must be listed in the same order as the order of the enabling properties. For example,

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

Measure error vector magnitude

Description

The Error Vector Magnitude `EVM` object is a measurement of modulator or demodulator performance in an impaired signal.

Construction

`H = comm.EVM` creates an error vector magnitude (EVM) System object, `H`. This object measures the amount of impairment in a modulated signal.

`H = comm.EVM(Name,Value)` creates an EVM 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

Normalization

EVM normalization method

Specify the normalization method that the object uses in the EVM calculation. Choose from `Average reference signal power` | `Average constellation power` | `Peak constellation power`. The default is `Average reference signal power`.

AverageConstellationPower

Average constellation power

Specify the average constellation power (in watts) that the object uses to normalize the EVM measurements. Set this property to a positive, real scalar value with a data type of double, single, or integer. This property applies when you set the `Normalization` property to `Average constellation power`. The default is `1`.

PeakConstellationPower

Peak constellation power

Specify the peak constellation power (in watts) that the object uses to normalize the EVM measurements. Set this property to a positive, real scalar value with a data type of double, single, or integer. This property applies when you set the `Normalization` property to `Peak constellation power`. The default is `1`.

MaximumEVMOutputPort

Enable maximum EVM measurement output

When you set this property to `true`, the `step` method outputs maximum EVM measurements. The default is `false`. The maximum EVM output is the maximum EVM value measured in the current input frame.

XPercentileEVMOutputPort

Enable *X*-percentile EVM measurement output

When you set this property to `true`, the step method outputs *X*-percentile EVM measurements. The default is `false`. After you set this property the *X*-percentile EVM measurements persist. These measurements are obtained based on all the input frames since the last reset.

XPercentileValue

X-percentile value

Specify the *X*-percentile value (in percent) that the object uses to calculate the *X*-th percentile of the EVM measurements. The default is `95`. The *X*-th percentile is the EVM value below which *X*% of all the computed EVM values lie. Set this property to a real scalar between `0` and `100`, inclusive. This property can have a data type of double, single, or integer, and applies when you set the `XPercentileEVMOutputPort` property to `true`.

SymbolCountOutputPort

Enable symbol count output

When you set this property to `true`, the step method outputs the number of accumulated symbols that the object uses to calculate the X-Percentile EVM measurements since the last reset. The default setting for this property is `false`. This property applies when you set the `XPercentileEVMOutputPort` property to true.

Methods

clone	Create EVM measurement object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of EVM measurement object
step	Measure error vector magnitude

Examples

Measure the EVM of a noisy 16-QAM modulated signal.

```
hMod = comm.RectangularQAMModulator(16);
hAWGN = comm.AWGNChannel('NoiseMethod',...
        'Signal to noise ratio (SNR)',...
        'SNR', 20, 'SignalPower', 10);

% Create an EVM object, output maximum and 90-percentile EVM
% measurements, and symbol count
    hEVM = comm.EVM('MaximumEVMOutputPort',true,...
            'XPercentileEVMOutputPort', true, 'XPercentileValue', 90,.
            'SymbolCountOutputPort', true);
 % Generate modulated symbols and add noise
    refsym = step(hMod, randi([0 15], 1000, 1));
    rxsym = step(hAWGN, refsym);
 % Calculate measurements
    [RMSEVM,MaxEVM,PercentileEVM,NumSym] = step(hEVM,refsym,rxsym)
```

For an additional example that uses this object, see the EVM Measurements for a 802.15.4 ZigBee System example. To open this exampleopen this example file, enter `edit EVMZigBeeDemo` at the MATLAB command line.

Algorithms

This object implements the algorithm, inputs, and outputs described on the EVM Measurement block reference page. The object properties correspond to the block parameters.

See Also

`comm.MER` | `comm.ACPR` | `comm.CCDF`

Purpose

Create EVM measurement object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `EVM` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.EVM.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `EVM` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.EVM.reset

Purpose Reset states of EVM measurement object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `EVM` object, `H`.

Purpose

Measure error vector magnitude

Syntax

```
RMSEVM = step(H,REFSYM,RXSYM)
[RMSEVM,MAXEVM] = step(H,REFSYM,RXSYM)
[RMSEVM,PEVM] = step(H,REFSYM,RXSYM)
[RMSEVM,NUMSYM] = step(H,REFSYM,RXSYM)
[RMSEVM,MAXEVM,PEVM,NUMSYM] = step(H,REFSYM,RXSYM)
```

Description

`RMSEVM = step(H,REFSYM,RXSYM)` outputs the root-mean-square EVM (in percent), `RMSEVM`, measured in the received signal, `RXSYM`, based on the reference signal, `REFSYM`. `REFSYM` and `RXSYM` inputs are complex column vectors of equal dimensions and data type. The data type can be double, single, signed integer, or signed fixed point with power-of-two slope and zero bias. The step method outputs the `RMSEVM` measurement based solely on the current input frame.

`[RMSEVM,MAXEVM] = step(H,REFSYM,RXSYM)` outputs the maximum EVM (in percent), `MAXEVM`, measured in the received signal, `RXSYM`, when you set the `MaximumEVMOutputPort` property to true. The step method outputs the `MAXEVM` measurement based on the reference signal, `REFSYM`. `MAXEVM` represents the worst-case EVM value measured in the current input frame.

`[RMSEVM,PEVM] = step(H,REFSYM,RXSYM)` outputs the percentile EVM (in percent), `PEVM`, measured in the received signal, `RXSYM`, when you set the `XPercentileEVMOutputPort` property to true. The step method outputs the `PEVM` measurement based on the reference signal, `REFSYM`. The object sets `PEVM` equal to a value just greater than the `XPercentileValue` percent of all the EVM values. For example, if you set the `XPercentileValue` property to 95, then 95% of all EVM measurements are below the `PEVM` value. The object calculates the persistent measurement `PEVM`, using all the input frames since the last reset.

`[RMSEVM,NUMSYM] = step(H,REFSYM,RXSYM)` outputs the number of symbols, `NUMSYM`, used to calculate the X-Percentile EVM measurements when you set the `SymbolCountOutputPort` property to true. All outputs of the object are of data type double. You can combine optional output arguments when you set their enabling properties. Optional outputs must be listed in the same order as the order of the enabling properties. For example,

```
[RMSEVM,MAXEVM,PEVM,NUMSYM] = step(H,REFSYM,RXSYM)
```

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

Demodulate using M-ary FSK method

Description

The `FSKDemodulator` object demodulates a signal that was modulated using the M-ary frequency shift keying method. The input is a baseband representation of the modulated signal. The input and output for this object are discrete-time signals.

Construction

`H = comm.FSKDemodulator` creates a demodulator System object, `H`. This object demodulates an M-ary frequency shift keying (M-FSK) signal using a noncoherent energy detector.

`H = comm.FSKDemodulator(Name,Value)` creates an M-FSK demodulator 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`).

`H = comm.FSKDemodulator(M,FREQSEP,RS,Name,Value)` creates an M-FSK demodulator object, `H`. This object has the `ModulationOrder` property set to `M`, the `FrequencySeparation` property set to `FREQSEP`, the `SymbolRate` property set to `RS`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of frequencies in modulated signal

Specify the number of frequencies in the modulated signal as a numeric, positive, integer scalar value that is a power of two. The default is `8`.

BitOutput

Output data as bits

Specify whether the output is groups of bits or integer values. The default is `false`.

When you set this property to `false`, the `step` method outputs a column vector of length equal to *N*/`SamplesPerSymbol`. *N* is the length of the input data vector to the step method. The elements of the output vector are integers between `0` and `ModulationOrder`–1. When you set this property to `true`, the `step` method outputs a column vector of length equal to **log2**(`ModulationOrder`)× (*N*/`SamplesPerSymbol`). The property's elements are bit representations of integers between `0` and `ModulationOrder`–1.

SymbolMapping

Symbol encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder`) bits to the corresponding symbol as one of `Binary | Gray`. The default is `Gray`.

When you set this property to `Gray`, the object uses Gray-coded ordering.

When you set this property to `Binary`, the object uses natural binary-coded ordering.

For either type of mapping, the object maps the highest frequency to the integer 0 and maps the lowest frequency to the integer M–1. In baseband simulation, the lowest frequency is the negative frequency with the largest absolute value.

FrequencySeparation

Frequency separation between successive tones

Specify the frequency separation between successive symbols in the modulated signal in Hertz as a positive, real scalar value. The default is `6` Hz.

SamplesPerSymbol

Number of samples per input symbol

Specify the number of samples per input symbol as a positive, integer scalar value. The default is `17`.

SymbolRate

Symbol duration

Specify the symbol rate in symbols per second as a positive, double-precision, real scalar value. The default is `100`. To avoid output signal aliasing, specify an output sampling rate, Fs = `SamplesPerSymbol`× `SymbolRate`, which is greater than `ModulationOrder` × `FrequencySeparation`. The symbol duration remain the same, regardless of whether the input is bits or integers.

OutputDataType

Data type of output

Specify the output data type as one of `logical | int8 | uint8 | int16 | uint16 | int32 | uint32 | double`. The default is `double`. The `logical` type is valid only when you set the `BitOutput` property to false and the `ModulationOrder` property to two. When you set the `BitOutput` property to true, the output data requires a type of `logical | double`.

Methods

clone	Create M-FSK demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of M-FSK demodulator object
step	Demodulate using M-ary FSK method

Examples

Modulate and demodulate a signal using 8-FSK modulation with a frequency separation of 100 Hz.

```
hMod = comm.FSKModulator(8, 100);
hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)','SNR',0);
hDemod = comm.FSKDemodulator(8, 100);
%Create an error rate calculator
hError = comm.ErrorRate;
for counter = 1:100
    % Transmit a 50-symbol frame
    data = randi([0 hMod.ModulationOrder-1],50,1);
    modSignal = step(hMod, data);
    noisySignal = step(hAWGN, modSignal);
    receivedData = step(hDemod, noisySignal);
    errorStats = step(hError, data, receivedData);
end
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
   errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-FSK Demodulator Baseband block reference page. The object properties correspond to the block parameters, except:

- The **Symbol set ordering** parameter corresponds to the `SymbolMapping` property.
- The `SymbolRate` property replaces the block sample rate capability.

See Also

`comm.FSKModulator` | `comm.CPFSKModulator` | `comm.CPFSKDemodulator`

Purpose Create M-FSK demodulator object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `FSKDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.FSKDemodulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.FSKDemodulator.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `FSKDemodulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.FSKDemodulator.reset

Purpose Reset states of M-FSK demodulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `FSKDemodulator` object, `H`.

Purpose

Demodulate using M-ary FSK method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` demodulates input data, `X`, with the FSK demodulator System object, `H`, and returns `Y`. `X` must be a double or single precision data type column vector of length equal to an integer multiple of the number of samples per symbol that you specify in the `SamplesPerSymbol` property. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

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

Modulate using M-ary FSK method

Description

The `FSKModulator` object modulates using the M-ary frequency shift keying method. The output is a baseband representation of the modulated signal.

Construction

`H = comm.FSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the M-ary frequency shift keying (M-FSK) method.

`H = comm.FSKModulator(Name,Value)` creates an M-FSK modulator 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`).

`H = comm.FSKModulator(M,FREQSEP,RS,Name,Value)` creates an M-FSK modulator object, `H`. This object has the `ModulationOrder` property set to `M`, the `FrequencySeparation` property set to `FREQSEP`, the `SymbolRate` property set to `RS`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of frequencies in modulated signal

Specify the number of frequencies in the modulated signal as a numeric positive integer scalar value that is a power of two. The default is `8`.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`.

When you set this property to `false`, the `step` method input requires a numeric (except single precision data type) column vector of integer values between `0` and `ModulationOrder`–1. In this case, the input vector can also be of data type logical if `ModulationOrder` equals `2`.

When you set this property to true, the step method input requires a double-precision or logical data type column vector of bit values. The length of this vector is an integer multiple of **log2**(ModulationOrder). This vector contains bit representations of integers between 0 and ModulationOrder–1.

SymbolMapping

Symbol encoding

Specify how the object maps an integer or group of **log2**(ModulationOrder) bits to the corresponding symbol as one of Binary | Gray. The default is Gray.

When you set this property to Gray, the object uses Gray-coded ordering.

When you set this property to Binary, the object uses natural binary-coded ordering. For either type of mapping, the object maps the highest frequency to the integer 0 and maps the lowest frequency to the integer M–1. In baseband simulation, the lowest frequency is the negative frequency with the largest absolute value.

FrequencySeparation

Frequency separation between successive tones

Specify the frequency separation between successive tones in the modulated signal in Hertz as a positive, real scalar value. The default is 6 Hz. To avoid output signal aliasing, specify an output sampling rate, Fs = SamplesPerSymbol/SymbolRate, which is greater than ModulationOrder multiplied by FrequencySeparation.

ContinuousPhase

Phase continuity

Specify if the phase of the output modulated signal is continuous or discontinuous. The default is true.

When you set this property to `true`, the modulated signal maintains continuous phase even when its frequency changes.

When you set this property to `false`, the modulated signal comprises portions of `ModulationOrder` sinusoids of different frequencies. In this case, a change in the input value can cause a discontinuous change in the phase of the modulated signal.

SamplesPerSymbol

Number of samples per output symbol

Specify the number of output samples that the object produces for each integer or binary word in the input as a positive, integer scalar value. The default is `17`.

SymbolRate

Symbol duration

Specify the symbol rate in symbols per second as a positive, double-precision, real scalar. The default is `100`. To avoid output signal aliasing, specify an output sampling rate, *Fs* = `SamplesPerSymbol`×`SymbolRate`, which is greater than `ModulationOrder` × `FrequencySeparation`. The symbol duration remain the same, regardless of whether the input is bits or integers.

OutputDataType

Data type of output

Specify the output data type as one of `double | single`. The default is `double`.

Methods

clone	Create M-FSK modulator object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of M-FSK modulator object
step	Modulate using M-ary FSK method

Examples

Modulate and demodulate a signal using 8-FSK modulation with a frequency separation of 100 Hz

```
hMod = comm.FSKModulator(8, 100);
hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)','SNR',0);
hDemod = comm.FSKDemodulator(8, 100);
%Create an error rate calculator
hError = comm.ErrorRate;
for counter = 1:100
    % Transmit a 50-symbol frame
    data = randi([0 hMod.ModulationOrder-1],50,1);
    modSignal = step(hMod, data);
    noisySignal = step(hAWGN, modSignal);
    receivedData = step(hDemod, noisySignal);
    errorStats = step(hError, data, receivedData);
end
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-FSK Modulator Baseband block reference page. The object properties correspond to the block parameters, except:

- The **Symbol set ordering** parameter corresponds to the `SymbolMapping` property.
- The `SymbolRate` property takes the place of the block sample rate capability.

See Also

`comm.FSKDemodulator` | `comm.CPFSKModulator`

Purpose Create M-FSK modulator object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `FSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.FSKModulator.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.FSKModulator.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `FSKModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.FSKModulator.reset

Purpose Reset states of M-FSK modulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `FSKModulator` object, `H`.

Purpose Modulate using M-ary FSK method

Syntax

```
Y = step(H,X)
```

Description `Y = step(H,X)` modulates input data, `X`, with the FSK modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit- valued column vector with numeric or logical data types. The length of output vector, `Y`, is equal to the number of input samples times the number of samples per symbol you specify in the `SamplesPerSymbol` 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.

---

comm.GardnerTimingSynchronizer

Purpose

Recover symbol timing phase using Gardner's method

Description

The `GardnerTimingSynchronizer` object recovers the symbol timing phase of the input signal using the Gardner method. This object implements a non-data-aided feedback method. Gardner timing synchronization is a non-data-aided feedback method that is independent of carrier phase recovery. The timing error detector that forms part of this object's algorithm requires at least two samples per symbol, one of which is the point at which the decision can be made.

Construction

`H = comm.GardnerTimingSynchronizer` creates a timing phase synchronizer System object, `H`. This object recovers the symbol timing phase of the input signal using the Gardner method.

`H = comm.GardnerTimingSynchronizer(Name,Value)` creates an Gardner timing synchronizer 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

SamplesPerSymbol

Number of samples representing each symbol

Specify the number of samples that represent each symbol in the input signal as an integer-valued scalar value greater than `1`. The default is `4`.

ErrorUpdateGain

Error update step size

Specify the step size for updating successive timing phase estimates as a positive real scalar value. The default is `0.05`. Typically, this number is less than 1/`SamplesPerSymbol`, which corresponds to a slowly varying timing phase. This property is tunable.

ResetInputPort

Enable synchronization reset input

Set this property to `true` to enable resetting the timing phase recovery process based on an input argument value. The default is `false`. When you set this property to `true`, you must specify a reset input value to the `step` method. When you specify a nonzero value as the reset input, the object restarts the timing phase recovery process. When you set this property to `false`, the object does not restart.

ResetCondition

Condition for timing phase recovery reset

Specify the conditions to reset the timing phase recovery process as one of `Never | Every frame`. The default is `Never`. When you set this property to `Never`, the phase recovery process never restarts. The object operates continuously, retaining information from one symbol to the next. When you set this property to `Every frame`, the timing phase recovery restarts at the start of each frame of data. In this case, the restart occurs each time the object calls the step method. This property applies when you set the `ResetInputPort` property to false.

Methods

clone	Create Gardner timing phase synchronizer object with same with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset	Reset states of Gardner timing phase synchronizer object
step	Recover symbol timing phase using Gardner's method

Examples

Recover timing phase using the Gardner method.

```
% Initialize some data
    L = 16; M = 8; numSymb = 100; snrdB = 30;
    R = 25; rollOff = 0.75; filtDelay = 3; g = 0.07; delay = 6.6498;
    % Design raised cosine filters
    txFiltSpec = fdesign.pulseshaping(L, 'Square root raised cosine', ...
                    'Nsym,Beta', 2*filtDelay, rollOff);
    txFilterDesign = design(txFiltSpec);
    txFilterDesign.Numerator = sqrt(L)*txFilterDesign.Numerator;
    % Create System objects
    hMod = comm.PSKModulator(M);
    hTxFilter = dsp.FIRInterpolator(L, txFilterDesign.Numerator);
    hDelay = dsp.VariableFractionalDelay('MaximumDelay', L);
    hChan = comm.AWGNChannel(...
                    'NoiseMethod', 'Signal to noise ratio (SNR)', ...
                    'SNR', snrdB, 'SignalPower', 1/L);
    hRxFilter = dsp.DigitalFilter('TransferFunction', 'FIR (all zeros)',
                    'Numerator', txFilterDesign.Numerator);
    hSync = comm.GardnerTimingSynchronizer('SamplesPerSymbol', L, ...
                    'ErrorUpdateGain', g);
% Generate random data
    data = randi([0 M-1], numSymb, 1);
% Modulate and filter transmitter data.
    modData = step(hMod, data);
    filterData = step(hTxFilter, modData);
% Introduce a random delay.
    delayedData = step(hDelay, filterData, delay);
% Add noise
    chData = step(hChan, delayedData);
% Filter receiver data.
```

```
    rxData = step(hRxFilter, chData);
% Estimate the delay from the received signal
    [~, phase] = step(hSync, rxData);
    fprintf(1, 'Actual Timing Delay: %f\n', delay);
    fprintf(1, 'Estimated Timing Delay: %f\n', phase(end));
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Gardner Timing Recovery block reference page. The object properties correspond to the block parameters, except:

The **Reset** parameter corresponds to the `ResetInputPort` and `ResetCondition` properties.

See Also

`comm.EarlyLateGateTimingSynchronizer` | `comm.MuellerMullerTimingSynchronizer`

comm.GardnerTimingSynchronizer.clone

Purpose

Create Gardner timing phase synchronizer object with same with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `GardnerTimingSynchronizer` 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)` method returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`.

comm.GardnerTimingSynchronizer.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.GardnerTimingSynchronizer.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `GardnerTimingSynchronizer` 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.

comm.GardnerTimingSynchronizer.release

Purpose

Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 Gardner timing phase synchronizer object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `GardnerTimingSynchronizer` object, `H`.

Purpose

Recover symbol timing phase using Gardner's method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Description

`[Y,PHASE] = step(H,X)` recovers the timing phase and returns the time-synchronized signal, `Y`, and the estimated timing phase, `PHASE`, for input signal `X`. The input `X` must be a double or single precision complex column vector. The length of `X` is N*K, where N is an integer greater than or equal to two and K is the number of symbols. The output, `Y`, is the signal value for each symbol, which you use to make symbol decisions. `Y` is a column vector of length K with the same data type as `X`.

`[Y,PHASE] = step(H,X,R)` restarts the timing phase recovery process when you input a reset signal, `R`, that is non-zero. `R` must be a logical or double scalar. This syntax applies 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.

Purpose

Demodulate using arbitrary QAM constellation

Description

The `GeneralQAMDemodulator` object demodulates a signal that was modulated using quadrature amplitude modulation. The input is a baseband representation of the modulated signal.

Construction

`H = comm.GeneralQAMDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using a general quadrature amplitude modulation (QAM) method.

`H = comm.GeneralQAMDemodulator(Name,Value)` creates a general QAM demodulator 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`).

`H = comm.GeneralQAMDemodulator(CONST,Name,Value)` creates a general QAM demodulator object, `H`. This object has the `Constellation` property set to `CONST`, and the other specified properties set to the specified values.

Properties

Constellation

Signal constellation

Specify the constellation points as a real or complex, double-precision data type vector. The default is

$\mathbf{exp}(2\times\pi\times 1i\times{}^{(0:7)}\!/_{8})$. The length of the vector determines the modulation order.

When you set the `BitOutput` property to `false`, the `step` method outputs a vector with integer values. These integers are between `0` and M–1, where M is the length of this property vector. The length of the output vector equals the length of the input signal.

When you set the `BitOutput` property to `true`, the output signal contains bits. For bit outputs, the size of the signal constellation requires an integer power of two and the output length is an integer multiple of the number of bits per symbol.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. The default is `false`.

When you set this property to `true` the `step` method outputs a column vector of bit values with length equal to **log2**(M) times the number of demodulated symbols, where M is the length of the signal constellation specified in the `Constellation` property. The length M determines the modulation order.

When you set this property to `false`, the `step` method outputs a column vector, of length equal to the input data vector. The vector contains integer symbol values between `0` and M–1.

DecisionMethod

Demodulation decision method

Specify the decision method the object uses as one of `Hard decision | Log-likelihood ratio | Approximate log-likelihood ratio`. The default is `Hard decision`. When you set the `BitOutput` property to false the object always performs hard decision demodulation. This property applies when you set the `BitOutput` property to true.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of `Property | Input port`. The default is `Property`. This property applies when you set the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`.

Variance

Noise variance

Specify the variance of the noise as a nonzero, real scalar value. The default is `1`. If this value is very small (i.e., SNR is very high), log-likelihood ratio (LLR) computations may yield Inf or -Inf. This result occurs because the LLR algorithm would compute the exponential of very large or very small numbers using finite-precision arithmetic. In such cases, using approximate LLR is recommended because its algorithm does not compute exponentials. This property applies when you set the `VarianceSource` property to `Property`. This property is tunable.

OutputDataType

Data type of output

Specify the output data type as one of `Full precision` | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`. The default is `Full precision` .

This property applies only when you set the `BitOutput` property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision` or `Approximate log-likelihood ratio`. In this case, when you set the `OutputDataType` property to `Full precision`, the output data type is the same as that of the input when the input data has a single or double-precision data type.

When the input data is of a fixed-point type, the output data type works as if you had set the `OutputDataType` property to `Smallest unsigned integer`.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in `Smallest unsigned integer` or `Full precision` mode.

When you set the `BitOutput` property to `true`, and the `DecisionMethod` property to `Hard Decision` the data type `logical` becomes a valid option.

When you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Approximate log-likelihood ratio` you may only set this property to `Full precision` | `Custom`.

If you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Log-likelihood ratio`, the output data has the same type as that of the input. In this case, that value can be only single or double precision.

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 of fixed-point numeric values

Specify the rounding method as one of `Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero`. The default is `Floor`. This property applies when the object is not in a full precision configuration. This property does not apply when you set `BitOutput` to true and `DecisionMethod` to `Log-likelihood ratio`.

OverflowAction

Action when fixed-point numeric values overflow

Specify the overflow action as one of `Wrap | Saturate`. The default is `Wrap`. This property applies when the object is not in a full precision configuration. This property does not apply when you set the `BitOutput` property to true and the `DecisionMethod` property to `Log-likelihood ratio`.

ConstellationDataType

Data type of signal constellation

Specify the constellation fixed-point data type as one of `Same word length as input | Custom`. The default is `Same word length as input`. This property does not apply when you set the `BitOutput` property to true and the `DecisionMethod` property to `Log-likelihood ratio`.

CustomConstellationDataType

Fixed-point data type of signal constellation

Specify the constellation fixed-point type as an unscaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `ConstellationDataType` property to `Custom`.

Accumulator1DataType

Data type of accumulator 1

Specify the accumulator 1 fixed-point data type as one of `Full precision | Custom`. The default is `Full precision`. This property applies when you set the `FullPrecisionOverride` property to false. This property does not apply when you set the `BitOutput` property to true and the `DecisionMethod` property to `Log-likelihood ratio`.

CustomAccumulator1DataType

Fixed-point data type of accumulator 1

Specify the accumulator 1 fixed-point type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `Accumulator1DataType` property to `Custom`.

ProductInputDataType

Data type of product

Specify the product input fixed-point data type as one of `Same as accumulator 1 | Custom`. The default is `Same as accumulator 1`. This property applies when you set the `FullPrecisionOverride` property to false, the `BitOutput` property to true and the `DecisionMethod` property to `Log-likelihood ratio`.

CustomProductInputDataType

Fixed-point data type of product

Specify the product input fixed-point type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` property to false and the `ProductInputDataType` property to `Custom`.

ProductOutputDataType

Data type of product output

Specify the product output fixed-point data type as one of `Full precision | Custom`. The default is `Full precision` . This property applies when you set the `FullPrecisionOverride` property to false, the `BitOutput` property to true and the `DecisionMethod` property to `Log-likelihood ratio`.

CustomProductOutputDataType

Fixed-point data type of product output

Specify the product output fixed-point type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` property to false and the `ProductOutputDataType` property to `Custom`.

Accumulator2DataType

Data type of accumulator 2

Specify the accumulator 2 fixed-point data type as one of `Full precision | Custom`. The default is `Full precision`. This property applies when you set the `FullPrecisionOverride` property to false, the `BitOutput` property to `true` and the `DecisionMethod` property to `Log-likelihood ratio`.

CustomAccumulator2DataType

Fixed-point data type accumulator 2

Specify the accumulator 2 fixed-point data type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` property to false and the `Accumulator2DataType` property to `Custom`.

Accumulator3DataType

Data type of accumulator 3

Specify the accumulator 3 fixed-point data type as one of `Full precision | Custom`. The default is `Full precision`. This property applies when you set the `FullPrecisionOverride` property to false, the `BitOutput` property to true and the `DecisionMethod` property to `Approximate log-likelihood ratio`.

CustomAccumulator3DataType

Fixed-point data type of accumulator 3

Specify the accumulator 3 fixed-point type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` property to false and the `Accumulator3DataType` property to `Custom`.

NoiseScalingInputDataType

Data type of noise-scaling input

Specify the noise-scaling input fixed-point data type as one of `Same as accumulator 3 | Custom`. The default is `Same as accumulator 3`. This property applies when you set the `FullPrecisionOverride` property to false, the `BitOutput` property to true and the `DecisionMethod` property to `Approximate log-likelihood ratio`.

CustomNoiseScalingInputDataType

Fixed-point data type of noise-scaling input

Specify the noise-scaling input fixed-point type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` property to false and the `NoiseScalingInputDataType` property to `Custom`.

InverseVarianceDataType

Data type of inverse noise variance

Specify the inverse noise variance fixed-point data type as one of `Same word length as input | Custom`. The default is `Same word length as input`. This property applies when you set the `BitOutput` property to true, the `DecisionMethod` property to `Approximate log-likelihood ratio`, and the `VarianceSource` property to `Property`.

CustomInverseVarianceDataType

Fixed-point data type of inverse noise variance

Specify the inverse noise variance fixed-point type as a `numerictype` object with a Signedness of Auto. The default is `numerictype([],16,8)`. This property applies when you set the `InverseVarianceDataType` property to `Custom`.

CustomOutputDataType

Data type of output

Specify the output fixed-point type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` property to false and the `OutputDataType` property to `Custom`.

Methods

clone	Create general QAM demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Demodulate using arbitrary QAM constellation

Examples

Modulate and demodulate data using an arbitrary three-point constellation.

```
% Setup a three point constellation
   c = [1 1i -1];
   hQAMMod = comm.GeneralQAMModulator(c);
   hAWGN = comm.AWGNChannel('NoiseMethod', ...
              'Signal to noise ratio (SNR)','SNR',15, 'SignalPower',
   hQAMDemod = comm.GeneralQAMDemodulator(c);

%Create an error rate calculator
   hError = comm.ErrorRate;
   for counter = 1:100
       % Transmit a 50-symbol frame
       data = randi([0 2],50,1);
       modSignal = step(hQAMMod, data);
```

```
        noisySignal = step(hAWGN, modSignal);
        receivedData = step(hQAMDemod, noisySignal);
        errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General QAM Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

comm.GeneralQAMModulator | comm.RectangularQAMDemodulator

Purpose Create general QAM demodulator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `GeneralQAMDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.GeneralQAMDemodulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.GeneralQAMDemodulator.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `GeneralQAMDemodulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.GeneralQAMDemodulator.step

Purpose

Demodulate using arbitrary QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

`Y = step(H,X)` demodulates the input data, `X`, with the general QAM demodulator System object, `H`, and returns `Y`. Input `X` must be a scalar or a column vector with double or single precision data type. When you set the `BitOutput` property to true and the `DecisionMethod` property to 'Log-likelihood ratio' the input data type must be single or double precision. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the `BitOutput` property to true, the `DecisionMethod` property to `Approximate log-likelihood ratio` or `Log-likelihood ratio`, and the `VarianceSource` 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

Modulate using arbitrary QAM constellation

Description

The `GeneralQAMModulator` object modulates using quadrature amplitude modulation. The output is a baseband representation of the modulated signal.

Construction

`H = comm.GeneralQAMModulator` creates a modulator System object, `H`. This object modulates the input signal using a general quadrature amplitude modulation (QAM) method.

`H = comm.GeneralQAMModulator(Name,Value)` creates a QAM modulator 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`).

`H = comm.GeneralQAMModulator(CONST,Name,Value)` creates a General QAM modulator object, `H`. This object has the `Constellation` property set to `CONST`, and the other specified properties set to the specified values.

Properties

Constellation

Signal constellation

Specify the constellation points as a vector of real or complex double-precision data type. The default is $\mathbf{exp}(2\times\pi\times 1i\times{}^{(0:7)}\!/_{8})$. The length of the vector determines the modulation order. The `step` method inputs requires integers between 0 and M–1, where M indicates the length of this property vector. The object maps an input integer m to the $(m+1)^{st}$ value in the Constellation vector.

OutputDataType

Data type of output

Specify the output data type as one of `double | single | Custom`. The default is `double`.

Fixed-Point Properties

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `OutputDataType` property to `Custom`.

Methods

clone	Create general QAM modulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Modulate using arbitrary QAM constellation

Examples

Modulate data using an arbitrary 3-point constellation. Then, visualize the data in a scatter plot

```
     hQAMMod = comm.GeneralQAMModulator;
 % Setup a three point constellation
     hQAMMod.Constellation = [1 1i -1];
     data = randi([0 2],100,1);
     modData = step(hQAMMod, data);
     scatterplot(modData)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General QAM Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.GeneralQAMDemodulator` | `comm.RectangularQAMModulator`

comm.GeneralQAMModulator.clone

Purpose

Create general QAM modulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `GeneralQAMModulator` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.GeneralQAMModulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `GeneralQAMModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Modulate using arbitrary QAM constellation

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` modulates input data, `X`, with the general QAM modulator System object, `H`. It returns the baseband modulated output, `Y`. The input must be an integer scalar or an integer-valued column vector. The data type of the input can be numeric or unsigned fixed point of word length **ceil**(**log2**(`ModulationOrder`)) (`fi` object).

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

Demodulate convolutionally encoded data mapped to arbitrary QAM constellation

Description

The `GeneralQAMTCMDemodulator` object uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using an arbitrary signal constellation.

Construction

`H = comm.GeneralQAMTCMDemodulator` creates a trellis-coded, general quadrature amplitude (QAM TCM) demodulator System object, `H`. This object demodulates convolutionally encoded data that has been mapped to an arbitrary QAM constellation.

`H = comm.GeneralQAMTCMDemodulator(Name,Value)` creates a general QAM TCM demodulator 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`).

`H = comm.GeneralQAMTCMDemodulator(TRELLIS,Name,Value)` creates a general QAM TCM demodulator object, `H`. This object has the `TrellisStructure` property set to `TRELLIS`, and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the `istrellis` function to check if a structure is a valid trellis structure. The default is the value that results from `poly2trellis([1 3], [1 0 0; 0 5 2])`.

TerminationMethod

Termination method of encoded frame

Specify the termination method as one of `Continuous | Truncated | Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object saves the internal state metric at the end of each frame. The next frame uses the same state metric. The object treats each traceback path independently. If the input signal contains only one symbol, use `Continuous` mode.

When you set this property to `Truncated`, the object treats each input vector independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state.

When you set this property to `Terminated`, the object treats each input vector independently, and the traceback path always starts and ends in the all-zeros state.

TracebackDepth

Traceback depth for Viterbi decoder

Specify the scalar, integer number of trellis branches to construct each traceback path. The default is `21`. The Traceback depth parameter influences the decoding accuracy and delay. The decoding delay indicates the number of zero symbols that precede the first decoded symbol in the output.

When you set the `TerminationMethod` property to `Continuous`, the decoding delay consists of `TracebackDepth` zero symbols or `TracebackDepth`$\times K$ zero bits for a rate K/N convolutional code.

When you set the `TerminationMethod` property to `Truncated` or `Terminated`, no output delay occurs and the traceback depth must be less than or equal to the number of symbols in each input vector.

ResetInputPort

Enable demodulator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to their initial conditions. This property applies when you set the `TerminationMethod` property to `Continuous`.

Constellation

Signal constellation

Specify a double- or single-precision complex vector. This vector lists the points in the signal constellation that were used to map the convolutionally encoded data. The constellation must be specified in set-partitioned order. See documentation for the General TCM Encoder block for more information on set-partitioned order. The length of the constellation vector must equal the number of possible input symbols to the convolutional decoder of the general QAM TCM demodulator object. This corresponds to 2^N for a rate K/N convolutional code. The default corresponds to a set-partitioned order for the points of an 8-PSK signal constellation. This value is expressed as

$\mathbf{exp}(2\times\pi\times j\times{}^{[0\quad 4\quad 2\quad 6\quad 1\quad 5\quad 3\quad 7]}\!/_{8})$.

OutputDataType

Data type of output

Specify output data type as one of `logical | double`. The default is `double`.

Methods

clone	Create general QAM TCM demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset	Reset states of the general QAM TCM demodulator object
step	Demodulate convolutionally encoded data mapped to arbitrary QAM constellation

Examples

Modulate and demodulate data using QAM TCM modulation with an arbitrary 4-point constellation.

```
% Define a trellis structure with binary inputs and 4-ary symbol outpu
    t = poly2trellis(7,[171 133]);
    const = exp(pi*1i*[1 2 3 6]/4);
    hMod = comm.GeneralQAMTCMModulator(t, 'Constellation', const);
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)', ...
                    'SNR',5, 'SignalPower',0.875);
    hDemod = comm.GeneralQAMTCMDemodulator(t, 'Constellation', const);
% Create an error rate calculator with delay in bits equal to Tracebac
    hError = comm.ErrorRate(...
           'ReceiveDelay', hDemod.TracebackDepth*log2(t.numInputSymbol
    for counter = 1:10
      data = randi([0 1],500,1);
      modSignal = step(hMod, data);
      noisySignal = step(hAWGN, modSignal);
      receivedData = step(hDemod, noisySignal);
      errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General TCM Decoder block reference page. The object properties correspond to the block parameters.

See Also

`comm.GeneralQAMTCMModulator` | `comm.RectangularQAMTCMDemodulator` | `comm.ViterbiDecoder`

comm.GeneralQAMTCMDemodulator.clone

Purpose

Create general QAM TCM demodulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `GeneralQAMTCMDemodulator` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.GeneralQAMTCMDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `GeneralQAMTCMDemodulator` 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.

comm.GeneralQAMTCMDemodulator.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the general QAM TCM demodulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `GeneralQAMTCMDemodulator` object, `H`.

comm.GeneralQAMTCMDemodulator.step

Purpose

Demodulate convolutionally encoded data mapped to arbitrary QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

`Y = step(H,X)` demodulates the general QAM modulated input data, `X`, and uses the Viterbi algorithm to decode the resulting demodulated convolutionally encoded bits. `X` must be a complex double or single precision column vector. The `step` method outputs a demodulated binary column data vector, `Y`. When the convolutional encoder represents a rate K/N code, the length of the output vector equals $K \times L$, where L is the length of the input vector, `X`.

`Y = step(H,X,R)` resets the decoder states of the general QAM TCM demodulator System object to the all-zeros state when you input a non-zero reset signal, `R`. `R` must be a double precision or logical scalar integer. This syntax applies 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.

Purpose

Convolutionally encode binary data and map using arbitrary QAM constellation

Description

The `GeneralQAMTCMModulator` object implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal. The object then maps the result to an arbitrary signal constellation. The Signal constellation property lists the signal constellation points in set-partitioned order.

Construction

`H = comm.GeneralQAMTCMModulator` creates a trellis-coded, general quadrature amplitude (QAM TCM) modulator System object, `H`. This object convolutionally encodes a binary input signal and maps the result using QAM modulation with a signal constellation specified in the Constellation property.

`H = comm.GeneralQAMTCMModulator(Name,Value)` creates a general QAM TCM modulator System 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`).

`H = comm.GeneralQAMTCMModulator(TRELLIS,Name,Value)` creates a general QAM TCM modulator System object, `H`. This object has the `TrellisStructure` property set to `TRELLIS`, and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the `istrellis` function to check if a structure is a valid trellis structure. The default is the result of `poly2trellis([1 3], [1 0 0; 0 5 2])`.

TerminationMethod

Termination method of encoded frame

Specify the termination method as one of `Continuous | Truncated | Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object retains the encoder states at the end of each input vector for use with the next input vector.

When you set this property to `Truncated`, the object treats each input vector independently. The encoder is reset to the all-zeros state at the start of each input vector.

When you set this property to `Terminated`, the object treats each input vector independently. For each input vector, the object uses extra bits to set the encoder to the all-zeros state at the end of the vector. For a rate K/N code, the `step` method outputs the vector with length $y = N \times {(L+S)}/{K}$, where S = *constraintLength*–1. In the case of multiple constraint lengths, S = **sum**(*constraintLength*(*i*)–1)). L represents the length of the input to the `step` method.

ResetInputPort

Enable modulator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to their initial conditions. This property applies when you set the `TerminationMethod` property to `Continuous`.

Constellation

Signal constellation

Specify a double- or single-precision complex vector that lists the points in the signal constellation that were used to map the convolutionally encoded data. You must specify the constellation in set-partitioned order. See documentation for the General TCM Encoder block for more information on set-partitioned order. The length of the constellation vector must equal the number of possible input symbols to the convolutional decoder of the general QAM TCM demodulator object. This corresponds to 2^N for a rate K/N convolutional code. The default corresponds to a set-partitioned order for the points of an 8-PSK signal constellation. This value is expressed

$\mathbf{exp}(2\times\pi\times j\times^{[0\quad 4\quad 2\quad 6\quad 1\quad 5\quad 3\quad 7]}\!/_{8})$.

OutputDataType

Data type of output

Specify the output data type as one of `double | single`. The default is `double`.

Methods

clone	Create general QAM TCM modulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the general QAM TCM modulator object
step	Convolutionally encode binary data and map using arbitrary QAM constellation

Examples

Modulate data using QAM TCM modulation with an arbitrary 4-point constellation.

```
% Create binary data.
    data = randi([0 1], 1000,1);
```

```
% Use the trellis structure with generating polynomial [171 133] and 4-pc
    t = poly2trellis(7,[171 133]);
    hMod = comm.GeneralQAMTCMModulator(t,...
                    'Constellation', exp(pi*1i*[1 2 3 6]/4));
% Modulate and plot the data
    modData = step(hMod, data);
    scatterplot(modData);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General TCM Encoder block reference page. The object properties correspond to the block parameters.

See Also

`comm.GeneralQAMTCMDemodulator` | `comm.GeneralQAMModulator` | `comm.PSKTCMModulator` | `comm.ConvolutionalEncoder`

Purpose Create general QAM TCM modulator object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `GeneralQAMTCMModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.GeneralQAMTCMModulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.GeneralQAMTCMModulator.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `tTF = isLocked(H)` returns the locked status, `TF` of the `GeneralQAMTCMModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.GeneralQAMTCMModulator.reset

Purpose Reset states of the general QAM TCM modulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `GeneralQAMTCMModulator` object, `H`.

Purpose

Convolutionally encode binary data and map using arbitrary QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

`Y = step(H,X)` convolutionally encodes and modulates the input data, `X`, and returns the encoded and modulated data, `Y`. `X` must be of data type numeric, logical, or unsigned fixed point of word length 1 (fi object). When the convolutional encoder represents a rate K/N code, the length of the input vector, `X`, must be $K \times L$, for some positive integer L. The `step` method outputs a complex column vector, `Y`, of length L.

`Y = step(H,X,R)` resets the encoder of the general QAM TCM modulator object to the all-zeros state when you input a non-zero reset signal, `R`. `R` must be a double precision or logical scalar integer. This syntax applies 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.

---

comm.GMSKDemodulator

Purpose

Demodulate using GMSK method and the Viterbi algorithm

Description

The `GMSKDemodulator` object uses a Viterbi algorithm to demodulate a signal that was modulated using the Gaussian minimum shift keying method. The input is a baseband representation of the modulated signal.

Construction

`H = comm.GMSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input Gaussian minimum shift keying (GMSK) modulated data using the Viterbi algorithm.

`H = comm.GMSKDemodulator(Name,Value)` creates a GMSK demodulator object, `H`. This object has 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

BitOutput

Output data as bits

Specify whether the output is groups of bits or integer values. The default is `false`.

When you set the `BitOutput` property to `false`, the `step` method outputs a column vector of length equal to *N*/`SamplesPerSymbol`. *N* is the length of the input signal, which is the number of input baseband modulated symbols. The elements of the output vector are `1` or `1`.

When you set the `BitOutput` property to `true`, the `step` method outputs a binary column vector of length equal to *N*/`SamplesPerSymbol` with bit values of `0` or `1`.

BandwidthTimeProduct

Product of bandwidth and symbol time of Gaussian pulse

Specify the product of bandwidth and symbol time for the Gaussian pulse shape as a real, positive scalar. The default `0.3`.

PulseLength

Pulse length

Specify the length of the Gaussian pulse shape in symbol intervals as a real positive integer. The default 4.

SymbolPrehistory

Symbol prehistory

Specify the data symbols used by the modulator prior to the first call to the `step` method. The default is `1`. This property requires a scalar or vector with elements equal to `-1` or `1`. If the value is a vector, its length must be one less than the value you set in the `PulseLength` property.

InitialPhaseOffset

Initial phase offset

Specify the initial phase offset of the input modulated waveform in radians as a real, numeric scalar value. The default is `0`.

SamplesPerSymbol

Number of samples per input symbol

Specify the expected number of samples per input symbol as a positive, integer scalar value. The default is `8`.

TracebackDepth

Traceback depth for Viterbi algorithm

Specify the number of trellis branches that the Viterbi algorithm uses to construct each traceback path as a positive, integer scalar value. The value of this property is also the output delay, and the number of zero symbols that precede the first meaningful demodulated symbol in the output. The default is `16`.

OutputDataType

Data type of output

Specify the output data type as one of `int8 | int16 | int32 | double`, when you set the `BitOutput` property to `false`.

When you set the BitOutput property to true, specify the output data type as one of logical | double. The default is double.

Methods

clone	Create GMSK demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the GMSK demodulator object
step	Demodulate using GMSK method and the Viterbi algorithm

Examples

Modulate and demodulate a signal using GMSK modulation with bit inputs and an initial phase offset of pi/4.

```
   hMod = comm.GMSKModulator('BitInput', true, ...
                   'InitialPhaseOffset', pi/4);
   hAWGN = comm.AWGNChannel('NoiseMethod', ...
                   'Signal to noise ratio (SNR)','SNR',0);
   hDemod = comm.GMSKDemodulator('BitOutput', true, ...
                   'InitialPhaseOffset', pi/4);
% Create an error rate calculator, account for the delay caused by the V
   hError = comm.ErrorRate('ReceiveDelay', hDemod.TracebackDepth);
   for counter = 1:100
     % Transmit 100 3-bit words
     data = randi([0 1],300,1);
     modSignal = step(hMod, data);
```

```
  noisySignal = step(hAWGN, modSignal);
  receivedData = step(hDemod, noisySignal);
  errorStats = step(hError, data, receivedData);
end
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the GMSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.GMSKModulator` | `comm.CPMModulator` | `comm.CPMDemodulator`

Purpose

Create GMSK demodulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `GMSKDemodulator` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.GMSKDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `GMSKDemodulator` 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.

comm.GMSKDemodulator.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the GMSK demodulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `GMSKDemodulator` object, `H`.

Purpose

Demodulate using GMSK method and the Viterbi algorithm

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` demodulates input data, `X`, with the GMSK demodulator object, `H`, and returns `Y`. `X` must be a double or single precision column vector with a length equal to an integer multiple of the number of samples per symbol you specify in the `SamplesPerSymbol` 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

Modulate using GMSK method

Description

The `GMSKModulator` object modulates using the Gaussian minimum shift keying method. The output is a baseband representation of the modulated signal.

Construction

`H = comm.GMSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the Gaussian minimum shift keying (GMSK) modulation method.

`H = comm.GMSKModulator(Name,Value)` creates a GMSK modulator object, `H`. This object has 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

BitInput

Assume input is bits

Specify whether the input is bits or integers. The default is `false`.

When you set the `BitInput` property to `false`, the `step` method input requires a double-precision or signed integer data type column vector with values of `-1` or `1`.

When you set the `BitInput` property to true, step method input requires a double-precision or logical data type column vector of 0s and 1s.

BandwidthTimeProduct

Product of bandwidth and symbol time of Gaussian pulse

Specify the product of the bandwidth and symbol time for the Gaussian pulse shape as a real, positive scalar value. The default is `0.3`.

PulseLength

Pulse length

Specify the length of the Gaussian pulse shape in symbol intervals as a real, positive integer. The default is `4`.

SymbolPrehistory

Symbol prehistory

Specify the data symbols the modulator uses prior to the first call to the `step` method in reverse chronological order. The default is `1`. This property requires a scalar or vector with elements equal to `-1` or `1`. If the value is a vector, then its length must be one less than the value in the `PulseLength` property.

InitialPhaseOffset

Initial phase offset

Specify the initial phase of the modulated waveform in radians as a real, numeric scalar value. The default is `0`.

SamplesPerSymbol

Number of samples per output symbol

Specify the upsampling factor at the output as a real, positive, integer scalar value. The default is `8`. The upsampling factor is the number of output samples that the `step` method produces for each input sample.

OutputDataType

Data type of output

Specify output data type as one of `double | single`. The default is `double`.

Methods

clone	Create GMSK modulator object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the GMSK modulator object
step	Modulate using GMSK method

Examples

Modulate and demodulate a signal using GMSK modulation with bit inputs and an initial phase offset of pi/4.

```
    hMod = comm.GMSKModulator('BitInput', true, 'InitialPhaseOffset',
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.GMSKDemodulator('BitOutput', true, ...
                    'InitialPhaseOffset', pi/4);
% Create an error rate calculator, account for the delay caused by the
    hError = comm.ErrorRate('ReceiveDelay', hDemod.TracebackDepth);
    for counter = 1:100
      % Transmit 100 3-bit words
      data = randi([0 1],300,1);
      modSignal = step(hMod, data);
      noisySignal = step(hAWGN, modSignal);
      receivedData = step(hDemod, noisySignal);
      errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the GMSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

comm.GMSKDemodulator | comm.CPMModulator | comm.CPMDemodulator

Purpose

Create GMSK modulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `GMSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.GMSKModulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.GMSKModulator.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `GMSKModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.GMSKModulator.reset

Purpose Reset states of the GMSK modulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `GMSKModulator` object, `H`.

Purpose

Modulate using GMSK method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` modulates input data, `X`, with the GMSK modulator object, `H`. It returns the baseband modulated output in `Y`. Depending on the `BitInput` property value, input `X` can be a double precision, signed integer, or logical column vector. The length of vector `Y` is equal to the number of input samples times the number of samples per symbol that you specify in the `SamplesPerSymbol` 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.

---

comm.GMSKTimingSynchronizer

Purpose

Recover symbol timing phase using fourth-order nonlinearity method

Description

The `GMSKTimingSynchronizer` object recovers the symbol timing phase of the input signal using a fourth-order nonlinearity method. This block implements a general non-data-aided feedback method. This timing synchronization is a non-data-aided feedback method that is independent of carrier phase recovery, but requires prior compensation for the carrier frequency offset. You can use this block for systems that use Gaussian minimum shift keying (GMSK) modulation.

Construction

`H = comm.GMSKTimingSynchronizer` creates a timing phase synchronizer System object, `H`. This object recovers the symbol timing phase of the GMSK input signal using a fourth-order nonlinearity method.

`H = comm.GMSKTimingSynchronizer(Name,Value)` creates a GMSK timing synchronizer 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

SamplesPerSymbol

Number of samples representing each symbol

Specify the number of samples that represent each symbol in the input signal as an integer-valued scalar value greater than `1`. The default is `4`.

ErrorUpdateGain

Error update step size

Specify the step size for updating successive timing phase estimates as a positive real scalar value. Typically, this number is less than 1/`SamplesPerSymbol`, which corresponds to a slowly varying timing phase. The default is `0.05`. This property is tunable.

ResetInputPort

Enable synchronization reset input

Set this property to `true` to enable resetting the timing phase recovery process based on an input argument value. The default is `false`.

When you set this property to `true`, you must specify a reset input value to the `step` method.

When you specify a nonzero value as the reset input, the object restarts the timing phase recovery process. When you set this property to `false`, the object does not restart.

ResetCondition

Condition for timing phase recovery reset

Specify the conditions to reset the timing phase recovery process as one of `Never | Every frame`. The default is `Never`.

When you set this property to `Never`, the phase recovery process never restarts. The object operates continuously, retaining information from one symbol to the next.

When you set this property to `Every frame`, the timing phase recovery restarts at the start of each frame of data. In this case, the restart occurs at each `step` method call. This property applies when you set the `ResetInputPort` property to `false`.

Methods

clone	Create GMSK timing phase synchronizer object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
reset	Reset states of GMSK timing phase synchronizer object
step	Recover symbol timing phase using fourth-order nonlinearity method

Examples

Recover timing phase of an MSK signal.

```
% Create System objects
   hMod = comm.GMSKModulator('BitInput', true, ...
                   'SamplesPerSymbol', 14);
   timingOffset = 0.2; % Actual timing offset
   hDelay = dsp.VariableFractionalDelay;
   hSync = comm.GMSKTimingSynchronizer('SamplesPerSymbol', 14, ...
                   'ErrorUpdateGain', 0.05);
   phEst = zeros(1, 10);
   for i = 1:51
     data = randi([0 1], 100, 1); % generate data
     modData = step(hMod, data); % modulate data

% data impaired by timing offset error
     impairedData = step(hDelay, modData, timingOffset*14);
     % perform timing phase recovery
     [y, phase] = step(hSync, impairedData);
     phEst(i) = phase(1)/14;
   end

   figure, plot(0.2*ones(1, 50));
   hold on; ylim([0 0.4])
   plot(phEst, 'r'); legend( 'original', 'estimated')
   title('Original and Estimated timing phases');
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the MSK-Type Signal Timing Recovery block reference page. The object properties correspond to the block parameters, except:

- The object corresponds to the MSK-Type Signal Timing Recovery block with the **Modulation type** parameter set to `GMSK`.
- The **Reset** parameter corresponds to the `ResetInputPort` and `ResetCondition` properties.

See Also

`comm.EarlyLateGateTimingSynchronizer` | `comm.MuellerMullerTimingSynchronizer`

comm.GMSKTimingSynchronizer.clone

Purpose

Create GMSK timing phase synchronizer object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `GMSKTimingSynchronizer` 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)` method returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`.

comm.GMSKTimingSynchronizer.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `GMSKTimingSynchronizer` 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.

comm.GMSKTimingSynchronizer.release

Purpose Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 GMSK timing phase synchronizer object

Syntax `reset(H)`

Description `reset(H)` resets the states for the `GMSKTimingSynchronizer` object `H`.

comm.GMSKTimingSynchronizer.step

Purpose

Recover symbol timing phase using fourth-order nonlinearity method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Description

`[Y,PHASE] = step(H,X)` performs timing phase recovery and returns the time-synchronized signal, `Y`, and the estimated timing phase, `PHASE`, for input signal `X`. `X` must be a double or single precision complex column vector.

`[Y,PHASE] = step(H,X,R)` restarts the timing phase recovery process when you input a reset signal, `R`, that is non-zero. `R` must be a logical or double scalar. This syntax applies 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.

Purpose

Generate Gold sequence

Description

The GoldSequence object generates a Gold sequence. Gold sequences form a large class of sequences that have good periodic cross-correlation properties.

Construction

H = comm.GoldSequence creates a Gold sequence generator System object, H. This object generates a pseudo-random Gold sequence.

H = comm.GoldSequence(Name,Value) creates a Gold sequence generator 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

FirstPolynomial

Generator polynomial for first preferred PN sequence

Specify the polynomial that determines the feedback connections for the shift register of the first preferred PN sequence generator. The default is [1 0 0 0 0 1 1]. You can specify the generator polynomial as a numeric, binary vector that lists the coefficients of the polynomial in descending order of powers. The first and last elements must equal 1, and the length of this vector requires a value of n+1, where n is the degree of the generator polynomial. Alternatively, you can specify the generator polynomial as a numeric vector containing the exponents of z for the nonzero terms of the polynomial in descending order of powers. The last entry must be 0. For example, [1 0 0 0 0 0 1 0 1] and [8 2 0] represent the same polynomial, $g(z) = z^8 + z^2 + 1$. The degree of the first generator polynomial must equal the degree of the second generator polynomial specified in the SecondPolynomial property.

FirstInitialConditions

Initial conditions for first PN sequence generator

Specify the initial conditions for the shift register of the first preferred PN sequence generator. The default is [0 0 0 0 0 1]. The initial conditions require a numeric, binary scalar, or a numeric, binary vector with length equal to the degree of the first generator polynomial specified in the FirstPolynomial property. If you set this property to a vector, each element of the vector corresponds to the initial value of the corresponding cell in the shift register. If you set this property to a scalar, the initial conditions of all shift register cells are the specified scalar value.

SecondPolynomial

Generator polynomial for second preferred PN sequence

Specify the polynomial that determines the feedback connections for the shift register of the second preferred PN sequence generator. The default is [1 1 0 0 1 1 1]. You can specify the generator polynomial as a binary, numeric vector that lists the coefficients of the polynomial in descending order of powers. The first and last elements must equal 1, and the length of this vector requires a value of n+1, where n is the degree of the generator polynomial. Alternatively, you can specify the generator polynomial as a numeric vector containing the exponents of z for the nonzero terms of the polynomial in descending order of powers. The last entry must be 0. For example, [1 0 0 0 0 0 1 0 1] and [8 2 0] represent the same polynomial, $g(z) = z^8 + z^2 + 1$. The degree of the second generator polynomial must equal the degree of the first generator polynomial specified in the FirstPolynomial property.

SecondInitialConditionsSource

Source of initial conditions for second PN sequence

Specify the source of the initial conditions that determines the start of the second PN sequence as one of Property | Input port. The default is Property. When you set this property to Property, you can specify the initial conditions as a scalar or binary vector using the SecondInitialConditions property. When you set this property to `Input port`, you specify the initial conditions as an input to the `step`method. The object accepts a binary scalar or a binary vector input. The length of the input must equal the degree of the generator polynomial that the `SecondPolynomial` property specifies.

SecondInitialConditions

Initial conditions for second PN sequence generator

Specify the initial conditions for the shift register of the second preferred PN sequence generator as a numeric, binary scalar, or as a numeric, binary vector. The length must equal the degree of the second generator polynomial. You set the second generator polynomial in the `SecondPolynomial` property.

When you set this property to a vector, each element of the vector corresponds to the initial value of the corresponding cell in the shift register. The default is `[0 0 0 0 0 1]`.

When you set this property to a scalar, the initial conditions of all shift register cells are the specified scalar value.

Index

Index of output sequence of interest

Specify the index of the output sequence of interest from the set of available sequences as a scalar integer. The default is `0`. The scalar integer must be in the range $[-2, 2^n-2]$, where n is the degree of the generator polynomials you specify in the `FirstPolynomial` and `SecondPolynomial` properties.

The index values `-2` and `-1` correspond to the first and second preferred PN sequences as generated by the `FirstPolynomial` and `SecondPolynomial`, respectively.

The set $G(u, v)$ of available Gold sequences is defined by $G(u,v) = \{u, v, (u \operatorname{xor} T^{v}), (u \operatorname{xor} T^{2v}), \ldots, (u \operatorname{xor} T^{((N-1)v)})\}$.In this case, T represents the operator that shifts vectors cyclically to the left by one place, and u, v represent the two preferred PN sequences.

Also, $G(u,v)$ contains N+2 Gold sequences of period N. You select the desired sequence from this set using the `Index` property.

Shift

Sequence offset from initial time

Specify the offset of the Gold sequence from its starting point as a numeric, integer scalar value that can be positive or negative.

The default is `0`. The Gold sequence has a period of $N = 2^n - 1$, where n is the degree of the generator polynomials specified in the `FirstPolynomial` and `SecondPolynomial` properties. The shift value is wrapped with respect to the sequence period.

VariableSizeOutput

Enable variable-size outputs

Set this property to true to enable an additional input to the step method. The default is false. When you set this property to true, the enabled input specifies the output size of the Gold sequence used for the step. The input value must be less than or equal to the value of the `MaximumOutputSize` property.

When you set this property to false, the `SamplesPerFrame` property specifies the number of output samples.

MaximumOutputSize

Maximum output size

Specify the maximum output size of the Gold sequence as a positive integer 2-element row vector. The second element of the vector must be 1. The default is [10 1].

This property applies when you set the `VariableSizeOutput` property to true.

SamplesPerFrame

Number of output samples per frame

Specify the number of Gold sequence samples that the `step` method outputs as a numeric, integer scalar value. The default is `1`. If you set this property to a value of M, then the `step` method outputs M samples of a Gold sequence with a period of $N = 2^n - 1$. The value of n represents the degree of the generator polynomials that you specify in the `FirstPolynomial` and `SecondPolynomial` properties.

ResetInputPort

Enable generator reset input

Set this property to `true` to enable an additional reset input to the `step` method. The default is `false`. This input resets the states of the two shift registers of the Gold sequence generator to the initial conditions specified in the `FirstInitialConditions` and `SecondInitialConditions` properties.

OutputDataType

Data type of output

Specify the output data type as one of `double | logical | Smallest unsigned integer`. The default is `double`.

You must have a Fixed-Point Designer user license to use this property in `Smallest unsigned integer` mode.

Methods

clone	Create Gold sequence generator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
reset	Reset states of Gold sequence generator object
step	Generate a Gold sequence

Examples

Get 10 samples of a Gold sequence of period $2^5 - 1$.

```
hgld = comm.GoldSequence('FirstPolynomial',[5 2 0],...
         'SecondPolynomial', [5 4 3 2 0],...
         'FirstInitialConditions', [0 0 0 0 1],...
         'SecondInitialConditions', [0 0 0 0 1],...
         'Index', 4, 'SamplesPerFrame', 10);
x = step(hgld)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Gold Sequence Generator block reference page. The object properties correspond to the block parameters.

See Also

`comm.KasamiSequence` | `comm.PNSequence`

Purpose Create Gold sequence generator object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `GoldSequence` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.GoldSequence.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.GoldSequence.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `GoldSequence` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.GoldSequence.reset

Purpose Reset states of Gold sequence generator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `GoldSequence` object, `H`.

Purpose

Generate a Gold sequence

Syntax

```
Y = step(H)
Y = step(H,RESET)
```

Description

`Y = step(H)` outputs a frame of the Gold sequence in column vector `Y`. Specify the frame length with the `SamplesPerFrame` property. The object uses two PN sequence generators to generate a preferred pair of sequences with period N = 2^n-1. Then the object XORs these sequences to produce the output Gold sequence. The value in n is the degree of the generator polynomials that you specify in the `FirstPolynomial` and `SecondPolynomial` properties.

`Y = step(H,RESET)` uses `RESET` as the reset signal when you set the `ResetInputPort` property to true. The data type of the `RESET` input must be double precision or logical. `RESET` can be a scalar value or a column vector with length equal to the number of samples per frame specified in the `SamplesPerFrame` property. When the `RESET` input is a non-zero scalar, the object resets to the initial conditions that you specify in the `FirstInitialConditions` and `SecondInitialConditions` properties. It then generates a new output frame. A column vector `RESET` input allows multiple resets within an output frame. A non-zero value at the ith element of the vector causes a reset at the ith output sample time.

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

Add white Gaussian noise to input signal with GPU

Description

The `GPU AWGNChannel` object adds white Gaussian noise to an input signal using a graphics processing unit (GPU).

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see “GPU Computing” in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see “Use gpuArray Data” in the Parallel Computing Toolbox documentation.

Construction

`H = comm.gpu.AWGNChannel` creates a GPU-based additive white Gaussian noise (AWGN) channel System object, `H`. This object adds white Gaussian noise to a real or complex input signal.

`H = comm.gpu.AWGNChannel(Name,Value)` creates a GPU-based AWGN channel 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).

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Properties

NoiseMethod

Method to specify noise level

Select the method to specify the noise level as one of `Signal to noise ratio (Eb/No)` | `Signal to noise ratio (Es/No)` | `Signal to noise ratio (SNR)` | `Variance`. The default is `Signal to noise ratio (Eb/No)`.

EbNo

Energy per bit to noise power spectral density ratio (Eb/No)

Specify the Eb/No ratio in decibels. Set this property to a numeric, real scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (Eb/No)`. The default is `10`. This property is tunable.

EsNo

Energy per symbol to noise power spectral density ratio (Es/No)

Specify the Es/No ratio in decibels. Set this property to a numeric, real scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (Es/No)`. The default is `10`. This property is tunable.

SNR

Signal to noise ratio (SNR)

Specify the SNR value in decibels. Set this property to a numeric, real scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (SNR)`. The default is `10`. This property is tunable.

BitsPerSymbol

Number of bits in one symbol

Specify the number of bits in each input symbol. You can set this property to a numeric, positive, integer scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (Eb/No)`. The default is `1` bit.

SignalPower

Input signal power in Watts

Specify the mean square power of the input signal in Watts. Set this property to a numeric, positive, real scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (Eb/No)`, `Signal to noise ratio (Es/No)` or `Signal to noise ratio (SNR)`. The default is `1` Watt. The object assumes a nominal impedance of `1` Ohm. This property is tunable.

SamplesPerSymbol

Number of samples per symbol

Specify the number of samples per symbol. Set this property to a numeric, positive, integer scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Signal to noise ratio (Eb/No)` or `Signal to noise ratio (Es/No)`. The default is 1 sample.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of `Property | Input port`. The default is `Property`. Set `VarianceSource` to `Input port` to specify the noise variance value via an input to the step method. Set `VarianceSource` to `Property` to specify the noise variance value using the `Variance` property. This property applies when you set the `NoiseMethod` property to `Variance`.

Variance

Noise variance

Specify the variance of the white Gaussian noise. You can set this property to a numeric, positive, real scalar or row vector with a length equal to the number of channels. This property applies when you set the `NoiseMethod` property to `Variance` and the `VarianceSource` property to `Property`. The default is `1`. This property is tunable.

RandomStream

Source of random number stream

Specify the source of random number stream. The only valid setting for this property is `Global stream`. The object generates the normally distributed random numbers from the current global random number stream.

Seed

Initial seed of `mt19937ar` random number stream

The GPU version of the AWGN Channel System object does not use this property.

Methods

clone	Create AWGN Channel object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Add white Gaussian noise to input signal

Algorithm

This object uses the same algorithm as the `comm.AWGNChannel` System object. See the Algorithms section of the `comm.AWGNChannel` help page for more details. The object properties correspond to the related block parameters, except that:

- This object uses `parallel.gpu.RandStream` to provide an interface for controlling the properties of one or more random number streams that the GPU uses. Usage is the same as `RandStream` with the following restrictions:
  - Only the `combRecursive` (MRG32K3A) generator is supported.
  - Only the `Inversion` normal transform is supported.
  - Setting the `substream` property is not allowed.

Enter help `parallel.gpu.RandStream` at the MATLAB command line for more information.

Examples

Add AWGN to an 8-PSK signal.

```
hMod = comm.PSKModulator;
modData = step(hMod,randi([0 hMod.ModulationOrder-1],1000,1));
hAWGN = comm.gpu.AWGNChannel('EbNo',15, 'BitsPerSymbol', ...
                log2(hMod.ModulationOrder));
channelOutput = step(hAWGN, modData);
```

```
% Visualize the noiseless and noisy data in scatter plots
scatterplot(modData)
scatterplot(channelOutput)
```

See Also

`comm.AWGNChannel`

comm.gpu.AWGNChannel.clone

Purpose Create AWGN Channel object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `GPU AWGN Channel` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `GPU AWGN Channel` 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.

comm.gpu.AWGNChannel.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 white Gaussian noise to input signal

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

`Y = step(H,X)` adds white Gaussian noise to input `X` and returns the result in `Y`. The input `X` can be a double or single precision data type scalar, vector, or matrix with real or complex values. The dimensions of input `X` determine single or multichannel processing. For an `M`-by-`N` matrix input, `M` represents the number of time samples per channel and `N` represents the number of channels. `M` and `N` can be equal to `1`. The object adds frames of length `M` of Gaussian noise to each of the `N` channels independently.

`Y = step(H,X,VAR)` uses input `VAR` as the variance of the white Gaussian noise. This applies when you set the `NoiseMethod` property to `Variance` and the `VarianceSource` property to `Input port`. Input `VAR` can be a positive scalar or row vector with a length equal to the number of channels. `VAR` must be of the same data type as 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.

---

comm.gpu.BlockDeinterleaver

Purpose

Restore original ordering of block interleaved sequence with GPU

Description

The `BlockDeinterleaver` System object restores the original ordering of a sequence that was interleaved using the block interleaver System object.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Construction

`H = comm.gpu.BlockDeinterleaver` creates a GPU-based block deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using the `BlockInterleaver` System object

H = comm.gpu.BlockDeinterleaver(Name,Value) creates a GPU-based block deinterleaver object, H, with the specified property name set to the specified value.

H = comm.gpu.BlockDeinterleaver(PERMVEC) creates a GPU-based block deinterleaver object, H, with the PermutationVector property set to PERMVEC.

Properties

PermutationVector

Permutation vector

Specify the mapping used to permute the input symbols as a column vector of integers. The default is [5;4;3;2;1]. The mapping is a vector where the number of elements is equal to the length, N, of the input to the step method. Each element must be an integer between 1 and N, with no repeated values.

Methods

clone	Create Block Deinterleaver object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Deinterleave input sequence

Algorithm

This object uses the same algorithm as the comm.BlockDeinterleaver System object. See Algorithms on the comm.BlockDeinterleaver help page for details.

Examples

Interleave and deinterleave data.

```
%Example 1: Interleave and deinterleave data
    hInt = comm.gpu.BlockInterleaver([3 4 1 2]');
    hDeInt = comm.gpu.BlockDeinterleaver([3 4 1 2]');
```

```
    data = randi(7, 4, 1);
    intData = step(hInt, data);
    deIntData = step(hDeInt, intData);
    % compare the original sequence, interleaved sequence,
    % and restored sequence
    [data, intData, deIntData]

%Example 2: Interleave and deinterleave data with random interleaver
    permVec = randperm(7)';  % Random permutation vector
    hInt = comm.gpu.BlockInterleaver(permVec);
    hDeInt = comm.gpu.BlockDeinterleaver(permVec);
    data = randi(9, 7, 1);
    intData = step(hInt, data);
    deIntData = step(hDeInt, intData);
    % compare the original sequence, interleaved sequence,
    % and restored sequence
    [data, intData, deIntData]
```

See Also

comm.gpu.BlockInterleaver | comm.BlockDeinterleaver

comm.gpu.BlockDeinterleaver.clone

Purpose

Create Block Deinterleaver object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `GPU Block Deinterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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.

comm.gpu.BlockDeinterleaver.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `GPU Block Deinterleaver` 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)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.gpu.BlockDeinterleaver.step

Purpose Deinterleave input sequence

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a block interleaver. The step method forms the output, `Y`, based on the mapping specified by the `PermutationVector` property as Output(PermutationVector(`k`))=Input(`k`), for `k` = 1:`N`, where `N` is the length of the permutation vector. The input `X` must be a column vector of the same length, `N`. The data type of `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `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

Create block interleaved sequence with GPU

Description

The `GPU BlockInterleaver` object permutes the symbols in the input signal using a graphics processing unit (GPU).

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Construction

`H = comm.gpu.BlockInterleaver` creates a GPU-based block interleaver System object, `H`. This object permutes the symbols in the input signal based on a permutation vector.

`H = comm.gpu.BlockInterleaver(Name,Value)` creates a GPU-based block interleaver object, `H`, with the specified property `Name` set to the specified `Value`.

`H = comm.gpu.BlockInterleaver(PERMVEC)` creates a GPU-based block deinterleaver object, `H`, with the `PermutationVector` property set to `PERMVEC`.

Properties

PermutationVector

Permutation vector

Specify the mapping used to permute the input symbols as a column vector of integers. The default is `[5;4;3;2;1]`. The mapping is a vector where the number of elements is equal to the length, `N`, of the input to the step method. Each element must be an integer between `1` and `N`, with no repeated values.

Methods

clone	Block Interleaver object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Permute input symbols using a permutation vector

Algorithm

The `GPU Block Interleaver` System object uses the same algorithm as the `comm.BlockInterleaver` System object. See Algorithms on the `comm.BlockInterleaver` help page for details.

Examples

Interleave and deinterleave data.

```
%Example 1: Interleave and deinterleave data
    hInt = comm.gpu.BlockInterleaver([3 4 1 2]');
    hDeInt = comm.gpu.BlockDeinterleaver([3 4 1 2]');
    data = randi(7, 4, 1);
    intData = step(hInt, data);
    deIntData = step(hDeInt, intData);
```

```
    % compare the original sequence, interleaved sequence,
    % and restored sequence
    [data, intData, deIntData]

%Example 2: Interleave and deinterleave data with random interleaver
    permVec = randperm(7)';  % Random permutation vector
    hInt = comm.gpu.BlockInterleaver(permVec);
    hDeInt = comm.gpu.BlockDeinterleaver(permVec);
    data = randi(9, 7, 1);
    intData = step(hInt, data);
    deIntData = step(hDeInt, intData);
    % compare the original sequence, interleaved sequence,
    % and restored sequence
    [data, intData, deIntData]
```

See Also

comm.gpu.BlockDeinterleaver | comm.BlockInterleaver

comm.gpu.BlockInterleaver.clone

Purpose

Block Interleaver object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `GPU Block Interleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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 Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `GPU Block Interleaver` 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.

comm.gpu.BlockInterleaver.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Permute input symbols using a permutation vector

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The step method forms the output `Y`, based on the mapping defined by the `PermutationVector` property as `Output(k)=Input(PermutationVector(k))`, for `k` = 1:`N`, where `N` is the length of the `PermutationVector` property. The input `X` must be a column vector of length `N`. The data type of `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `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.

---

comm.gpu.ConvolutionalEncoder

Purpose

Convolutionally encode binary data with GPU

Description

The `GPU ConvolutionalEncoder` object encodes a sequence of binary input vectors to produce a sequence of binary output vectors.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Construction

`H = comm.gpu.ConvolutionalEncoder` creates a System object, `H`, that convolutionally encodes binary data.

`H = comm.gpu.ConvolutionalEncoder(Name,Value)` creates a convolutional encoder 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`).

`H = comm.gpu.ConvolutionalEncoder(TRELLIS,Name,Value)` creates a convolutional encoder object, `H`. This object has the `TrellisStructure` property set to `TRELLIS`, and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the convolutional code. The default is the result of `poly2trellis(7, [171 133])`. Use the `istrellis` function to check if a structure is a valid trellis structure.

TerminationMethod

Termination method of encoded frame

Specify how the encoded frame is terminated as one of `Continuous | Truncated | Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object retains the encoder states at the end of each input vector for use with the next input vector.

When you set this property to `Truncated`, the object treats each input vector independently and resets its states to the all-zeros state.

When you set this property to `Terminated`, the object treats each input vector independently. For each input vector, the object uses extra bits to set the encoder states to the all-zeros state at the end of the vector. For a rate K/N code, the `step` method outputs a vector with length $N \times {}^{(L+S)}\!/_{K}$, where S = *constraintLength*–1. In the case of multiple constraint lengths, S = **sum**(*constraintLength*(*i*)–1)). L is the length of the input to the `step` method.

ResetInputPort

Enable encoder reset input

You cannot reset this encoder object using an input port. The only valid property setting is `false`.

DelayedResetAction

Delay output reset

You cannot reset this encoder object using an input port. The only valid property setting is `false`.

InitialStateInputPort

You cannot set the initial state of this encoder object. The only valid property setting is `false`.

FinalStateOutputPort

You cannot output the final state of this encoder object. The only valid property setting is `false`.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as one of `None | Property`. The default is `None`. When you set this property to `None` the object does not apply puncturing. When you set this property to `Property`, the object punctures the code. This puncturing is based on the puncture pattern vector that you specify in the `PuncturePattern` property. This property applies when you set the `TerminationMethod` property to `Continuous` or `Truncated`.

PuncturePattern

Puncture pattern vector

Specify the puncture pattern that the object uses to puncture the encoded data as a column vector. The default is `[1; 1; 0; 1; 0; 1]`. The vector contains `1`s and `0`s, where `0` indicates a punctured, or excluded, bit. This property applies when you set the `TerminationMethod` property to `Continuous` or `Truncated` and the `PuncturePatternSource` property to `Property`.

NumFrames

Number of independent frames present in the input and output data vectors.

Specify the number of independent frames contained in a single data input/output vector. The default value of this property is `1`. The objects segments the input vector into `NumFrames` segments and encodes them independently. The output contains `NumFrames` encoded segments. This property is applicable when you set the `TerminationMethod` to `Terminated` or `Truncated`.

Methods

clone	Create convolutional encoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the convolutional encoder object
step	Convolutionally encode binary data

Examples

8-PSK-Modulation With Convolutional Encoding

Transmit a Convolutionally Encoded, 8-PSK-Modulated Bit Stream Through an AWGN Channel.

Create a GPU-based Convolutional Encoder System object.

```
hConEnc = comm.gpu.ConvolutionalEncoder;
```

Create a GPU-based PSK Modulator System object that accepts a bit input signal.

```
hMod = comm.gpu.PSKModulator('BitInput',true);
```

Create a GPU-based AWGN Channel System object with a signal-to-noise ratio of seven.

```
hChan = comm.gpu.AWGNChannel('NoiseMethod', ...
         'Signal to noise ratio (SNR)',...
         'SNR',7);
```

Create a GPU-based PSK Demodulator System object that outputs a column vector of bit values.

```
hDemod = comm.gpu.PSKDemodulator('BitOutput',true);
```

Create a GPU-based Viterbi Decoder System object that accepts an input vector of hard decision values, which are zeros or ones.

```
hDec = comm.gpu.ViterbiDecoder('InputFormat','Hard');
```

Create an Error Rate System object that ignores 3 data samples before makings comparisons. The received data lags behind the transmitted data by 34 samples.

```
hError = comm.ErrorRate('ComputationDelay',3,'ReceiveDelay', 34);
```

Run the simulation by using the step method to process data.

```
for counter = 1:20
  data = randi([0 1],30,1);
  encodedData = step(hConEnc, gpuArray(data));
  modSignal = step(hMod, encodedData);
  receivedSignal = step(hChan, modSignal);
  demodSignal = step(hDemod, receivedSignal);
  receivedBits = step(hDec, demodSignal);
  errors = step(hError, data, gather(receivedBits));
end
```

Display the errors.

```
disp(errors)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Encoder block reference page. The object properties correspond to the block parameters.

See Also

comm.gpu.ViterbiDecoder | comm.gpu.ConvolutionalDeinterleaver | comm.gpu.ConvolutionalInterleaver | comm.ConvolutionalEncoder

comm.gpu.ConvolutionalEncoder.clone

Purpose Create convolutional encoder object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `GPU ConvolutionalEncoder` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.gpu.ConvolutionalEncoder.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `GPU ConvolutionalEncoder` 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.

comm.gpu.ConvolutionalEncoder.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)` Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the convolutional encoder object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `GPU ConvolutionalEncoder` object, `H`.

comm.gpu.ConvolutionalEncoder.step

Purpose Convolutionally encode binary data

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` encodes the binary data, `X`, using the convolutional encoding that you specify in the `TrellisStructure` property. It returns the encoded data, `Y`. Both `X` and `Y` are column vectors of data type single, double, or logical. When the convolutional encoder represents a rate K/N code, the length of the input vector equals $K \times L$, for a positive integer, L. The step method sets the length of the output vector, `Y`, to $L \times N$.

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

Permute input symbols using shift registers with GPU

Description

The `GPU ConvolutionalInterleaver` object permutes the symbols in the input signal using a graphics processing unit (GPU). Internally, this class uses a set of shift registers.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Construction

`H = comm.gpu.ConvolutionalInterleaver` creates a GPU-based convolutional interleaver System object, `H`. This object permutes the symbols in the input signal using a set of shift registers.

`H = comm.gpu.ConvolutionalInterleaver(Name,Value)` creates a GPU-based convolutional interleaver System 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`).

`H = comm.gpu.ConvolutionalInterleaver(M,B,IC)` creates a GPU-based convolutional interleaver System object `H`, with the `NumRegisters` property set to `M`, the `RegisterLengthStep` property set to `B`, and the `InitialConditions` property set to `IC`. `M`, `B`, and `IC` are value-only arguments. To specify a value-only argument, you must also specify all preceding value-only arguments.

Properties

NumRegisters

Number of internal shift registers

Specify the number of internal shift registers as a scalar, positive integer. The default is `6`.

RegisterLengthStep

Number of additional symbols that fit in each successive shift register

Specify the number of additional symbols that fit in each successive shift register as a positive, scalar integer. The default is `2`. The first register holds zero symbols.

InitialConditions

Initial conditions of shift registers

Specify the values that are initially stored in each shift register as a numeric scalar or vector. You do not need to specify a value for the first shift register, which has zero delay. The default is `0`. The value of the first element of this property is unimportant because the first shift register has zero delay. If you set this property to a scalar, then all shift registers, except the first one, store the same specified value. If you set it to a column vector with length equal to the value of the `NumRegisters` property, then the *i*-th shift register stores the *i*-th element of the specified vector.

Methods

clone	Create convolutional interleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the convolutional interleaver object
step	Permute input symbols using shift registers

Examples

Interleave and deinterleave random data

Interleave and deinterleave random data. Then, compare the original sequence, interleaved sequence and restored sequence.

Create a GPU-based Convolutional Interleaver with three internal shift registers capable of fitting two additional symbols. The initial value stored in each shift register is `[-1 -2 -3]`.

```
hInt = comm.gpu.ConvolutionalInterleaver('NumRegisters', 3, ...
    'RegisterLengthStep', 2, ...
    'InitialConditions', [-1 -2 -3]');
```

Create a GPU-based Convolutional Deinterleaver with three internal shift registers capable of fitting two additional symbols. The initial value stored in each shift register is `[-1 -2 -3]`.

```
hDeInt = comm.gpu.ConvolutionalDeinterleaver('NumRegisters', 3, ...
    'RegisterLengthStep', 2, ...
```

```
    'InitialConditions', [-1 -2 -3]');
```

Copy numeric data to the GPU.

```
data = gpuArray((0:20)');
```

Run the simulation by using the step method to process data.

```
intrlvData = step(hInt, data);
deintrlvData = step(hDeInt, intrlvData);
```

Compare the original sequence, interleaved sequence and restored sequence.

```
[data, intrlvData, deintrlvData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Interleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.ConvolutionalInterleaver` | `comm.gpu.ConvolutionalDeinterleaver`

Purpose

Create convolutional interleaver object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `GPU ConvolutionalInterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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.

comm.gpu.ConvolutionalInterleaver.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.gpu.ConvolutionalInterleaver.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `GPU ConvolutionalInterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.gpu.ConvolutionalInterleaver.reset

Purpose Reset states of the convolutional interleaver object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `GPU ConvolutionalInterleaver` object, `H`.

Purpose

Permute input symbols using shift registers

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The input X must be a column vector. The data type can be of type `double`, `single`, `uint32`, `int32`, or `logical`. `Y` has the same data type as `X`. The convolutional interleaver object uses a set of N shift registers, where N is the value specified by the `NumRegisters` property. The object sets the delay value of the k^{th} shift register to the product of (k-1) and the `RegisterLengthStep RegisterLengthStep` property value. With each new input symbol, a commutator switches to a new register and the new symbol shifts in while the oldest symbol in that register shifts out. When the commutator reaches the N^{th} register and the next new input occurs, it returns to the first register.

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

Restore ordering of symbols using shift registers with GPU

Description

The `GPU ConvolutionalDeinterleaver` object recovers a signal that was interleaved using the GPU-based convolutional interleaver object. The parameters in the two blocks should have the same values.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Construction

`H = comm.gpu.ConvolutionalDeinterleaver` creates a GPU-based convolutional deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using a convolutional interleaver.

`H = comm.gpu.ConvolutionalDeinterleaver(Name,Value)` creates a GPU-based convolutional deinterleaver System 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`).

`H = comm.gpu.ConvolutionalDeinterleaver(M,B,IC)` creates a convolutional deinterleaver System object `H`, with the `NumRegisters` property set to `M`, the `RegisterLengthStep` property set to `B`, and the `InitialConditions` property set to `IC`. `M`, `B`, and `IC` are value-only arguments. To specify a value-only argument, you must also specify all preceding value-only arguments.

Properties

NumRegisters

Number of internal shift registers

Specify the number of internal shift registers as a scalar, positive integer. The default is `6`.

RegisterLengthStep

Number of additional symbols that fit in each successive shift register

Specify the number of additional symbols that fit in each successive shift register as a positive, scalar integer. The default is `2`. The first register holds zero symbols.

InitialConditions

Initial conditions of shift registers

Specify the values that are initially stored in each shift register (except the first shift register, which has zero delay) as a numeric scalar or vector. The default is `0`. If you set this property to a scalar, then all shift registers, except the first one, store the same specified value. If you set it to a column vector with length equal to the value of the `NumRegisters`property, then the *i*-th shift register stores the *i*-th element of the specified vector. The value of the first element of this property is unimportant, since the first shift register has zero delay.

Methods

clone	Create convolutional deinterleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the convolutional deinterleaver object
step	Permute input symbols using shift registers

Examples

Interleave and Deinterleave Random Data

Interleave and deinterleave random data. Then, compare the original sequence, interleaved sequence and restored sequence.

Create a GPU-based Convolutional Interleaver with three internal shift registers capable of fitting two additional symbols. The initial value stored in each shift register is `[-1 -2 -3]`.

```
hInt = comm.gpu.ConvolutionalInterleaver('NumRegisters', 3, ...
    'RegisterLengthStep', 2, ...
    'InitialConditions', [-1 -2 -3]');
```

Create a GPU-based Convolutional Deinterleaver with three internal shift registers capable of fitting two additional symbols. The initial value stored in each shift register is `[-1 -2 -3]`.

```
hDeInt = comm.gpu.ConvolutionalDeinterleaver('NumRegisters', 3, ...
```

```
    'RegisterLengthStep', 2, ...
    'InitialConditions', [-1 -2 -3]');
```

Copy numeric data to the GPU.

```
data = gpuArray((0:20)');
```

Run the simulation by using the step method to process data.

```
intrlvData = step(hInt, data);
deintrlvData = step(hDeInt, intrlvData);
```

Compare the original sequence, interleaved sequence and restored sequence.

```
[data, intrlvData, deintrlvData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Deinterleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.ConvolutionalDeinterleaver` | `comm.gpu.ConvolutionalInterleaver`

comm.gpu.ConvolutionalDeinterleaver.clone

Purpose Create convolutional deinterleaver object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `GPU ConvolutionalDeinterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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.

comm.gpu.ConvolutionalDeinterleaver.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.gpu.ConvolutionalDeinterleaver.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `GPU ConvolutionalDeinterleaver` 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.

comm.gpu.ConvolutionalDeinterleaver.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)` Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Permute input symbols using shift registers

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a convolutional interleaver and returns `Y`. The input `X` must be a column vector. The data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The convolutional deinterleaver object uses a set of `N` shift registers, where `N` represents the value specified by the `NumRegisters` property. The object sets the delay value of the k^{th} shift register to the product of (k-1) and the `RegisterLengthStep` property value. With each new input symbol, a commutator switches to a new register and the new symbol shifts in while the oldest symbol in that register shifts out. When the commutator reaches the *N*th register and the next new input occurs, it returns to the first register.

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.

---

comm.gpu.ConvolutionalDeinterleaver.reset

Purpose Reset states of the convolutional deinterleaver object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `GPU ConvolutionalDeinterleaver` object, `H`.

Purpose

Decode binary low-density parity-check data with GPU

Description

The `GPU LDPCDecoder` object decodes a binary low-density parity-check code using a graphics processing unit (GPU).

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Construction

`h = comm.gpu.LDPCDecoder` creates a GPU-based binary low-densitycreates a GPU-based LDPC decoder object, `h`, with the ParityCheckMatrix property set to PARITY. parity-check (LDPC) decoder System object,`h`. This object performs LDPC decoding based on the specified parity-check matrix, where the object does not assume any patterns in the parity-check matrix.

`h = comm.gpu.LDPCDecoder('PropertyName','ValueName')` creates a GPU-based LDPC decoder object, `h`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`'PropertyName1'`,`'PropertyValue1'`,...,`'PropertyNameN'`,`'PropertyValueN'`).

`h = comm.gpu.LDPCDecoder(PARITY)` creates a GPU-based LDPC decoder object, `h`, with the ParityCheckMatrix property set to PARITY.

Properties

ParityCheckMatrix

Parity-check matrix

Specify the parity-check matrix as a binary valued sparse matrix with dimension (N-by-K) by N, where $N > K > 0$. This property accepts numeric or logical data types. The upper bound for the value of N is (2^{31})-1. The default is the parity-check matrix of the half-rate LDPC code from the DVB-S.2 standard, which is the result of dvbs2ldpc(1/2).

OutputValue

Select output value format

Specify the output value format as one of `Information part | Whole codeword`. The default is `Information part`. When you set this property to `Information part`, the output contains only the message bits and is a multiple of K length column vector, assuming an (N-by-K)xK parity check matrix. When you set this property to `Whole codeword`, the output contains the codeword bits and is an N element column vector.

DecisionMethod

Decision method

Specify the decision method used for decoding as one of `Hard decision | Soft decision`. The default is `Hard decision`. When you set this property to `Hard decision`, the output is decoded bits of logical data type. When you set this property to `Soft decision`, the output is log-likelihood ratios of single or double data type.

IterationTerminationCondition

Condition for iteration termination

Specify the condition to stop the decoding iterations as one of `Maximum iteration count` | `Parity check satisfied`. The default is `Maximum iteration count`. When you set this property to `Maximum iteration count`, the object will iterate for the number of iterations you specify in the `MaximumIterationCount` property. When you set this property to `Parity check satisfied`, the object will determine if the parity checks are satisfied after each iteration and stops if all parity checks are satisfied.

MaximumIterationCount

Maximum number of decoding iterations

Specify the maximum number of iterations the object uses as an integer valued numeric scalar. The default is `50`. This applies when you set the `IterationTerminationCondition` property to `Maximum iteration count`.

NumIterationsOutputPort

Output number of iterations performed

Set this property to true to output the actual number of iterations the object performed. The default is false.

FinalParityChecksOutputPort

Output final parity checks

Set this property to true to output the final parity checks the object calculated. The default is false.

Methods

clone	Create GPU LDPC Decoder object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Process inputs using the object algorithm

Algorithm

The `GPU LDPC Decoder` System object uses the same algorithm as the LDPC Decoder block. See Decoding Algorithm for details.

Examples

Transmit an LDPC-encoded, QPSK-modulated bit stream through an AWGN channel, then demodulate, decode, and count errors.

```
hEnc = comm.LDPCEncoder;
hMod = comm.PSKModulator(4, 'BitInput',true);
hChan = comm.AWGNChannel(...
        'NoiseMethod','Signal to noise ratio (SNR)','SNR',1);
hDemod = comm.PSKDemodulator(4, 'BitOutput',true,...
        'DecisionMethod','Approximate log-likelihood ratio', ...
        'Variance', 1/10^(hChan.SNR/10));
hDec = comm.gpu.LDPCDecoder;
hError = comm.ErrorRate;
for counter = 1:10
  data           = logical(randi([0 1], 32400, 1));
  encodedData    = step(hEnc, data);
  modSignal      = step(hMod, encodedData);
  receivedSignal = step(hChan, modSignal);
  demodSignal    = step(hDemod, receivedSignal);
  receivedBits   = step(hDec, demodSignal);
  errorStats     = step(hError, data, receivedBits);
end
fprintf('Error rate       = %1.2f\nNumber of errors = %d\n', ...
```

```
        errorStats(1), errorStats(2))
```

See Also

`comm.LDPCEncoder` | `comm.LDPCDecoder`

comm.gpu.LDPCDecoder.clone

Purpose

Create GPU LDPC Decoder object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a GPU `LDPCDecoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `ACPR` 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.

comm.gpu.LDPCDecoder.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)` releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Process inputs using the object algorithm

Syntax

```
Y = step(H,X)
[Y,NUMITER] = step(H,X)
[Y,PARITY] = step(H,X)
```

Description

`Y = step(H,X)` decodes input codeword, *X*, using an LDPC code that is based on an (*N-K*) x *N* parity-check matrix. You specify the parity-check matrix in the `ParityCheckMatrix` property. The input *X* must be a column vector of type double or single. Each element is the log-likelihood ratio for a received bit (more likely to be 0 if the log-likelihood ratio is positive). This System object is capable of decoding multiple frames of input data simultaneously. The length of the input *X* must be a multiple of *N*. The first *K* elements of every *N* elements correspond to the information part of a codeword. The decoded data output vector, *Y*, contains either only the message bits or the whole code word(s), based on the value of the `OutputValue` property.

`[Y,NUMITER] = step(H,X)` returns the actual number of iterations the object performed when you set the `NumIterationsOutputPort` property to true. The step method outputs NUMITER as a double scalar.

`[Y,PARITY] = step(H,X)` returns final parity checks the object calculated when you set the `FinalParityChecksOutputPort` property to true. The step method outputs PARITY as a logical vector of length (*N-K*).

You can combine optional output arguments when you set their enabling properties. Optional outputs must be listed in the same order as the order of the enabling properties. For example,

```
[Y,NUMITER,PARITY] = step(H,X)
```

Calling `step` on an object puts that object into a locked state. When locked, you cannot change non-tunable properties or any input characteristics (size, data type and complexity) without reinitializing (unlocking and relocking) the object.

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

Demodulate using M-ary PSK method with GPU

Description

The `GPU PSKDemodulator` object demodulates an input signal using the M-ary phase shift keying (M-PSK) method.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Construction

`H = comm.gpu.PSKDemodulator` returns a GPU-based demodulator System object, `H`. This object demodulates the input signal using the M-ary phase shift keying (M-PSK) method.

`H = comm.gpu.PSKDemodulator(Name,Value)` creates a GPU-based M-PSK demodulator object, `H`, with the specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN)

H = comm.gpu.PSKDemodulator(M,PHASE,Name,Value) creates a GPU-based M-PSK demodulator object, H, with the ModulationOrder property set to M, the PhaseOffset property set to PHASE and other specified property names set to the specified values. M and PHASE are value-only arguments. To specify a value-only argument, you must also specify all preceding value-only arguments. You can specify name-value pair arguments in any order.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar. The default is 8.

PhaseOffset

Phase of zeroth point of constellation

Specify the phase offset of the zeroth point of the constellation, in radians, as a real scalar. The default is π/8.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. When you set this property to true, the step method outputs a column vector of bit values with length equal to log2(ModulationOrder) times the number of demodulated symbols. When you set this property to false, the step method outputs a column vector, with a length equal to the input data vector that contains integer symbol values between 0 and ModulationOrder-1. The default is false.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of log2(ModulationOrder) bits to the corresponding symbol as Binary | Gray | Custom. The default is Gray. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the integer m (0 ≤ m ≤`ModulationOrder`-1) maps to the complex value. This value is represented as exp(j*`PhaseOffset` + j*2*pi*m/`ModulationOrder`). When you set this property to `Custom`, the object uses the signal constellation defined in the `CustomSymbolMapping` property.

CustomSymbolMapping

Custom constellation encoding

Specify a custom constellation symbol mapping vector. The default is `0:7`. This property must be a row or column vector of size `ModulationOrder` with unique integer values in the range [0, `ModulationOrder`-1]. The values must be of data type double. The first element of this vector corresponds to the constellation point at an angle of 0 + `PhaseOffset`, with subsequent elements running counterclockwise. The last element corresponds to the constellation point at an angle of -π/`ModulationOrder` + `PhaseOffset`. This property applies when you set the `SymbolMapping` property to Custom.

DecisionMethod

Demodulation decision method

Specify the decision method that the object uses as one of `Hard decision | Log-likelihood ratio | Approximate log-likelihood ratio`. The default is `Hard decision`. When you set `DecisionMethod` to false, the object always performs hard decision demodulation. This property applies when you set the `BitOutput` property to true.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of `Property | Input port`. The default is `Property`. This property applies when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`.

Variance

Specify the variance of the noise as a positive, real scalar. The default is `1`. If this value is very small (i.e., SNR is very high), then log-likelihood ratio (LLR) computations may yield Inf or -Inf. This occurs because the LLR algorithm computes the exponential value of very large or very small numbers using finite precision arithmetic. In such cases, use approximate LLR is recommended because its algorithm does not compute exponentials. This property applies when you set the `BitOutput` property to true, the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, and the `VarianceSource` property to `Property`. This property is tunable.

OutputDataType

Data type of output

When you set this property to `Full precision`, the output signal inherits its data type from the input signal.

Methods

clone	Create PSK demodulator object with same property values
constellation	Calculate or plot ideal signal constellation
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Demodulate using M-ary PSK method

Algorithm

The `GPU PSK Demodulator` System object uses the same algorithm as the `comm.PSKDemodulator` Communications System Toolbox object. See Decoding Algorithm for details.

Examples

Transmit an LDPC-encoded, QPSK-modulated bit stream through an AWGN channel. Then demodulate, decode, and count errors.

16-PSK Modulation and Demodulation

Transmit an LDPC-encoded, QPSK-modulated bit stream through an AWGN channel.

Create a GPU-based PSK Modulator System object.

```
hMod = comm.gpu.PSKModulator(16, 'PhaseOffset',pi/16);
```

Create a GPU-based AWGN Channel System object with a signal-to-noise ratio of 15.

```
hAWGN = comm.gpu.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (SNR)','SNR',15);
```

Create a GPU-based PSK Demodulator System object.

```
 hDemod = comm.gpu.PSKDemodulator(16, 'PhaseOffset',pi/16);
```

Create an error rate calculator System object.

```
hError = comm.ErrorRate;
```

Transmit a frame of data containing 50 symbols.

```
for counter = 1:100
data = gpuArray.randi([0 hMod.ModulationOrder-1], 50, 1);
```

Run the simulation by using the step method to process data.

```
modSignal = step(hMod, data);
noisySignal = step(hAWGN, modSignal);
receivedData = step(hDemod, noisySignal);
```

```
errorStats = step(hError, gather(data), gather(receivedData));
end
```

Compute the error rate results.

```
fprintf('Error rate = %f\nNumber of errors = %d\n',...
    errorStats(1), errorStats(2))
```

See Also

comm.PSKDemodulator | comm.gpu.PSKModulator

Purpose Create PSK demodulator object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `GPU PSK Demodulator` object, `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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.

comm.gpu.PSKDemodulator.constellation

Purpose

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.gpu.PSKDemodulator

Create a `comm.gpu.PSKDemodulator` System object, and then calculate its ideal signal constellation.

Create a `comm.gpu.PSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.gpu.PSKDemodulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.gpu.PSKDemodulator

Create a `comm.gpu.PSKDemodulator` System object, and then plot the ideal signal constellation.

Create a `comm.gpu.PSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.gpu.PSKDemodulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `GPU PSK Demodulator` 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.

comm.gpu.PSKDemodulator.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Demodulate using M-ary PSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

`Y = step(H,X)` demodulates data, `X`, with the `GPU PSK Demodulator` System object, `H`, and returns `Y`. Input `X` must be a scalar or a column vector with double- or single- precision data type. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the `BitOutput` property to true, the `DecisionMethod` property to `Approximate log-likelihood ratio` or `Log-likelihood ratio`, and the `VarianceSource` property to `Input port`. The data type of input `VAR` must be double or single precision.

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.

---

comm.gpu.PSKModulator

Purpose

Modulate using M-ary PSK method with GPU

Description

The `GPU PSKModulator` object modulates a signal using the M-ary phase shift keying method implemented on a graphics processing unit (GPU). The input is a baseband representation of the modulated signal. The input and output for this object are discrete-time signals. This object accepts a scalar-valued or column vector input signal.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Construction

`H = comm.gpu.PSKModulator` returns a GPU-based demodulator System object, `H`. This object modulates the input signal using the M-ary phase shift keying (M-PSK) method with soft decision using the approximate log-likelihood ratio algorithm.

H = comm.gpu.PSKModulator(Name,Value) creates a GPU-based M-PSK modulator 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)

H = comm.gpu.PSKModulator(M,PHASE,Name,Value) creates a GPU-based M-PSK modulator object, H, with the ModulationOrder property set to M, the PhaseOffset property set to PHASE and other specified property Names set to the specified Values. M and PHASE are value-only arguments. To specify a value-only argument, you must also specify all preceding value-only arguments. You can specify name-value pair arguments in any order.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar. The default is 8.

PhaseOffset

Phase of zeroth point of constellation

Specify the phase offset of the zeroth point of the constellation, in radians, as a real scalar. The default is π/8.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is false. When you set this property to true, the step method input must be a column vector of bit values whose length is an integer multiple of log2(ModulationOrder). This vector contains bit representations of integers between 0 and ModulationOrder-1. The input data type can be numeric or logical. When you set the BitInput property to false, the step method input must be a column vector of integer symbol values between 0 and ModulationOrder-1. The data type of the input must be numeric.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of log2(`ModulationOrder`) bits to the corresponding symbol as one of `Binary | Gray | Custom`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the integer m (0 ≤ m ≤ ModulationOrder-1) maps to the complex value exp(j*PhaseOffset + j*2*pi*m/ModulationOrder). When you set this property to `Custom`, the object uses the signal constellation defined in the `CustomSymbolMapping` property.

CustomSymbolMapping

Custom constellation encoding

Specify a custom constellation symbol mapping vector. This property must be a row or column vector of size `ModulationOrder` with unique integer values in the range [0, `ModulationOrder`-1]. The values must be of data type double. The first element of this vector corresponds to the constellation point at an angle of 0 + `PhaseOffset`, with subsequent elements running counterclockwise. The last element corresponds to the constellation point at an angle of -π/`ModulationOrder` + `PhaseOffset`. This property applies when you set the `SymbolMapping` property to Custom. The default is `0:7`.

OutputDataType

Data type of output

Specify the output data type as one of `double | single`. The default is `double`.

Methods

clone	Create PSK Modulator object with same property values
constellation	Calculate or plot ideal signal constellation

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Modulate using M-ary PSK method with GPU

Algorithm

The GPU PSK Modulator System object supports floating-point and integer input data types. This object uses the same algorithm as the comm.PSKModulator System object. See the Algorithms section of the comm.PSKModulator help page for details.

Examples

Modulate data using 16-PSK modulation and then visualize the data using a scatter plot.

```
% Create binary data for 24, 4-bit symbols
data = randi([0 1],96,1);
% Create a 16-PSK modulator System object with bits as inputs
% and Gray-coded signal constellation
hModulator = comm.gpu.PSKModulator(16,'BitInput',true);
% Change the phase offset to pi/16
hModulator.PhaseOffset = pi/16;
% Modulate and plot the data
modData = step(hModulator, data);
scatterplot(modData)
```

See Also

comm.PSKDemodulator

comm.gpu.PSKModulator.clone

Purpose Create PSK Modulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `GPU PSK Modulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.gpu.PSKModulator

Create a `comm.gpu.PSKModulator` System object, and then calculate its ideal signal constellation.

Create a `comm.gpu.PSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.gpu.PSKModulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.gpu.PSKModulator

Create a `comm.gpu.PSKModulator` System object, and then plot the ideal signal constellation.

Create a `comm.gpu.PSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.gpu.PSKModulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

comm.gpu.PSKModulator.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `GPU PSK Modulator` 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)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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
Modulate using M-ary PSK method with GPU

Syntax
`Y = step(H,X)`

Description
`Y = step(H,X)` modulates the input data, `X`, using the GPU-based PSK modulator System object, `H`. The object returns the baseband modulated output `Y`. Depending upon the value of the `BitInput` property, input `X` can be an integer or bit-valued column vector with numeric or logical data types.

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 input signal using parallel concatenation decoding with GPU

Description

The `GPU Turbo Decoder` System object decodes the input signal using a parallel concatenated decoding scheme. This scheme uses the *a-posteriori* probability (APP) decoder as the constituent decoder. Both constituent decoders use the same trellis structure and algorithm.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Construction

`H = comm.gpu.TurboDecoder` creates a GPU-based turbo decoder System object, `H`. This object uses the *a-posteriori* probability (APP) constituent decoder to iteratively decode the parallel-concatenated convolutionally encoded input data.

H = comm.gpu.TurboDecoder(Name, Value) creates a GPU-based turbo decoder object, H, with the specified property name set to the specified value. Name must appear inside single quotes (''). You can specify several name-value pair arguments in any order as Name1,Value1, ,NameN,ValueN.

H = comm.gpu.TurboDecoder(TRELLIS, INTERLVRINDICES, NUMITER) creates a GPU-based turbo decoder object, H. In this object, the TrellisStructure property is set to TRELLIS, the InterleaverIndices property set to INTERLVRINDICES, and the NumIterations property set to NUMITER.

Properties

TrellisStructure

Trellis structure of constituent convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the constituent convolutional code. The default is the result of poly2trellis(4, [13 15], 13). Use the istrellis function to check if a structure is a valid trellis structure.

InterleaverIndicesSource

Source of interleaver indices

Specify the source of the interleaver indices. The only valid setting for this property is Property.

InterleaverIndices

Interleaver indices

Specify the mapping used to permute the input bits at the encoder as a column vector of integers. The default is (64:-1:1).'.. This mapping is a vector with the number of elements equal to the length, *L*, of the output of the step method. Each element must be an integer between 1 and *L*, with no repeated values.

Algorithm

Decoding algorithm

Specify the decoding algorithm. This object implements true *a posteriori* probability decoding. The only valid setting is `True APP`.

NumScalingBits

Number of scaling bits

The GPU version of the Turbo Decoder does not use this property.

NumIterations

Number of decoding iterations

Specify the number of decoding iterations used for each call to the `step` method. The default is `6`. The object iterates and provides updates to the log-likelihood ratios (LLR) of the uncoded output bits. The output of the step method is the hard-decision output of the final LLR update.

NumFrames

Number of independent frames present in the input and output data vectors.

Specify the number of independent frames that a single data input/output vector contains. The default value of this property is `1`. This object segments the input vector into `NumFrames` segments and decodes the segments independently. The output contains `NumFrames` decoded segments.

Methods

clone	Create Turbo Decoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
reset	Reset states of the turbo decoder object
step	Decode input signal using parallel concatenated decoding scheme

Examples

Transmit and decode using turbo coding

Transmit turbo-encoded blocks of data over a BPSK-modulated AWGN channel. Then, decode using an iterative turbo decoder and display errors.

Define a noise variable, establish a frame length of 256, and use the random stream property so that the results are repeatable.

```
noiseVar = 4; frmLen = 256;
s = RandStream('mt19937ar', 'Seed', 11);
intrlvrIndices = randperm(s, frmLen);
```

Create a Turbo Encoder System object. The trellis structure for the constituent convolutional code is poly2trellis(4, [13 15 17], 13). The `InterleaverIndices` property specifies the mapping the object uses to permute the input bits at the encoder as a column vector of integers.

```
hTEnc = comm.TurboEncoder('TrellisStructure', poly2trellis(4, ...
      [13 15 17], 13), 'InterleaverIndices', intrlvrIndices);
```

Create a BPSK Modulator System object.

```
hMod = comm.BPSKModulator;
```

Create an AWGN Channel System object.

```
hChan = comm.AWGNChannel('NoiseMethod', 'Variance', 'Variance', ...
      noiseVar);
```

Create a GPU-Based Turbo Decoder System object. The trellis structure for the constituent convolutional code is poly2trellis(4, [13 15 17], 13). The `InterleaverIndicies` property specifies the mapping the object uses to permute the input bits at the encoder as a column vector of integers.

```
hTDec = comm.gpu.TurboDecoder('TrellisStructure', poly2trellis(4, ...
      [13 15 17], 13), 'InterleaverIndices', intrlvrIndices, ...
      'NumIterations', 4);
```

Create an Error Rate System object.

```
hError = comm.ErrorRate;
```

Run the simulation by using the step method to process data.

```
for frmIdx = 1:8
 data = randi(s, [0 1], frmLen, 1);
 encodedData = step(hTEnc, data);
 modSignal = step(hMod, encodedData);
 receivedSignal = step(hChan, modSignal);
```

Convert the received signal to log-likelihood ratios for decoding.

```
receivedBits  = step(hTDec, (-2/(noiseVar/2))*real(receivedSignal));
```

Compare original the data to the received data and then calculate the error rate results.

```
errorStats = step(hError, data, receivedBits);
end
fprintf('Error rate = %f\nNumber of errors = %d\nTotal bits = %d\n',
errorStats(1), errorStats(2), errorStats(3))
```

Algorithms

This object implements the inputs and outputs described on the Turbo Decoder block reference page. The object properties correspond to the block parameters.

See Also

`comm.TurboEncoder` | `comm.TurboDecoder`

Purpose

Create Turbo Decoder object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `GPU Turbo Decoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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.

comm.gpu.TurboDecoder.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(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 from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

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.

comm.gpu.TurboDecoder.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `TurboDecoder` 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)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

comm.gpu.TurboDecoder.reset

Purpose Reset states of the turbo decoder object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `GPU TurboDecoder` object, `H`.

Purpose

Decode input signal using parallel concatenated decoding scheme

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` decodes the input data, `X`, using the parallel concatenated convolutional coding scheme. You specify this scheme using the `TrellisStructure` and `InterleaverIndices` properties. It returns the binary decoded data, `Y`. Both `X` and `Y` are column vectors of double-precision data type. When the constituent convolutional code represents a rate 1/N code, the step method sets the length of the output vector, `Y`, to (`M`-2*numTails)/(2*`N`-1). `M` represents the input vector length and `numTails` is given by log2(`TrellisStructure.numStates`)*N. The output length, `L`, is the same as the length of the interleaver indices.

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 convolutionally encoded data using Viterbi algorithm with GPU

Description

The `GPU ViterbiDecoder` System object decodes input symbols to produce binary output symbols using a graphics processing unit (GPU). This object processes variable-size signals; however, variable-size signals cannot be applied for erasure inputs.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" in the Parallel Computing Toolbox documentation.

A GPU-based System object accepts typical MATLAB arrays or objects that you create using the gpuArray class as an input to the step method. GPU-based System objects support input signals with double- or single-precision data types. The output signal inherits its datatype from the input signal.

- If the input signal is a MATLAB array, then the output signal is also a MATLAB array. In this case, the System object handles data transfer between the CPU and GPU.
- If the input signal is a gpuArray, then the output signal is also a gpuArray. In this case, the data remains on the GPU. Therefore, when the object is given a gpuArray, calculations take place entirely on the GPU and no data transfer occurs. Invoking the step method with gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Use gpuArray Data" in the Parallel Computing Toolbox documentation.

Construction

`H = comm.gpu.ViterbiDecoder` creates a Viterbi decoder System object, `H`. This object uses the Viterbi algorithm to decode convolutionally encoded input data.

`H = comm.gpu.ViterbiDecoder(Name,Value)` creates a Viterbi decoder 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.

H = comm.gpu.ViterbiDecoder(TRELLIS,Name,Value) creates a Viterbi decoder object, H, with the TrellisStructure property set to TRELLIS, and other specified property Names set to the specified Values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the istrellis function to check if a structure is a valid trellis structure. This object supports rate 1/2, 1/3 and 1/4 trellises from simple feedforward encoders. The default value is the result of poly2trellis(7, [171 133]).

InputFormat

Input format

Specify the format of the input to the decoder as one of Unquantized | Hard | Soft. The default is Unquantized.

When you set this property to Unquantized, the input must be a real vector of double or single precision unquantized soft values. The object considers negative numbers to be ones and positive numbers to be zeros. When you set this property to Hard, the input must be a vector of hard decision values, which are zeros or ones. The data type of the inputs can be double precision or single precision. When you set this property to Soft, the input must be a vector of quantized soft values represented as integers between 0 and 2^SoftInputWordLength-1. The data type of the inputs can be double precision or single precision.

SoftInputWordLength

Soft input word length

Specify the number of bits used to represent each quantized soft input value as a positive, integer scalar. This property applies when you set the `InputFormat` property to `Soft`. The default is `4` bits.

InvalidQuantizedInputAction

Action when input values are out of range

The only valid setting is `Ignore` which ignores out of range inputs.

TracebackDepth

Traceback depth

Specify the number of trellis branches used to construct each traceback path as a positive, integer scalar less than or equal to 256. The traceback depth influences the decoding accuracy and delay. The number of zero symbols that precede the first decoded symbol in the output represent a decoding delay. When you set the `TerminationMethod` property to `Continuous`, the decoding delay consists of `TracebackDepth` zero symbols, or `TracebackDepth` zero bits for a rate 1/N convolutional code. When you set the `TerminationMethod` property to `Truncated` or `Terminated`, there is no output delay and `TracebackDepth` must be less than or equal to the number of symbols in each input. If the code rate is 1/2, a typical traceback depth value is about five times the constraint length of the code. The default is `34`.

TerminationMethod

Termination method of encoded frame

Specify `TerminationMethod` as one of `Continuous | Truncated | Terminated`. The default is `Continuous`. In `Continuous` mode, the object saves its internal state metric at the end of each frame for use with the next frame. The object treats each traceback path independently. Select `Continuous` mode when the input signal contains only one symbol. In `Truncated` mode, the object treats each frame independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state. In

Terminated mode, the object treats each frame independently, and the traceback path always starts and ends in the all-zeros state.

ResetInputPort

Enable decoder reset input

Set this property to true to enable an additional step method input. When the reset input is a non-zero value, the object resets the internal states of the decoder to initial conditions. This property applies when you set the TerminationMethod property to Continuous. The default is false.

DelayedResetAction

Delay output reset

Delaying the output reset is not supported. The only valid setting is false.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as one of None | Property. The default is None. When you set this property to None the object assumes no puncturing. Set this property to Property to decode punctured codewords based on a puncture pattern vector specified via the PuncturePattern property.

PuncturePattern

Puncture pattern vector

Specify puncture pattern used to puncture the encoded data. The default is [1; 1; 0; 1; 0; 1]. The puncture pattern is a column vector of ones and zeros, where the zeros indicate where to insert dummy bits. The puncture pattern must match the puncture pattern used by the encoder. This property applies when you set the PuncturePatternSource property to Property.

ErasuresInputPort

Enable erasures input

Erasures are not supported. The only valid setting is false.

OutputDataType

Data type of output

The only valid setting is `Full precision` which makes the output data type match the input data type.

NumFrames

Number of independent frames present in the input and output data vectors.

Specify the number of independent frames contained in a single data input/output vector. The input vector will be segmented into `NumFrames` segments and decoded independently. The output will contain `NumFrames` decoded segments. The default value of this property is `1`. This property is applies when you set the `TerminationMethod` is set to `Terminated` or `Truncated`.

Methods

clone	Create Viterbi Decoder object with same property values
info	Display information about GPU-based Viterbi Decoder object
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the GPU-based Viterbi Decoder modulator object
step	Decode convolutionally encoded data using Viterbi algorithm

Examples

Transmit a convolutionally encoded 8-DPSK-modulated bit stream through an AWGN channel. Then, demodulate, decode using a Viterbi decoder, and count errors.

```
hConEnc = comm.ConvolutionalEncoder;
hMod = comm.DPSKModulator('BitInput',true);
hChan = comm.gpu.AWGNChannel('NoiseMethod', ...
            'Signal to noise ratio (SNR)', 'SNR',10);
hDemod = comm.DPSKDemodulator('BitOutput',true);
hDec = comm.gpu.ViterbiDecoder('InputFormat','Hard');
% Delay in bits is TracebackDepth times the number of
% bits per symbol
   delay = hDec.TracebackDepth*...
              log2(hDec.TrellisStructure.numInputSymbols);
hError = comm.ErrorRate('ComputationDelay',3,'ReceiveDelay',delay);
    for counter = 1:20
      data = randi([0 1],30,1);
      encodedData = step(hConEnc, data);
      modSignal = step(hMod, encodedData);
      receivedSignal = step(hChan, modSignal);
      demodSignal = step(hDemod, receivedSignal);
      receivedBits = step(hDec, demodSignal);
      errorStats = step(hError, data, receivedBits);
    end
 fprintf('Error rate = %f\nNumber of errors = %d\n', ...
 errorStats(1), errorStats(2))
```

References

[1] Fettweis, G., H. Meyr. "Feedforward Architecture for Parallel Viterbi Decoding," *Journal of VLSI Signal Processing*, Vol. 3, June 1991.

See Also

`comm.ViterbiDecoder`

comm.gpu.ViterbiDecoder.clone

Purpose

Create Viterbi Decoder object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a GPU `Viterbi Decoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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

Display information about GPU-based Viterbi Decoder object

Syntax

`S = info(OBJ)`

Description

`S = info(OBJ)` returns a structure, `S`, containing characteristic information for the System object, `OBJ`. If `OBJ` has no characteristic information, `S` is empty. If `OBJ` has characteristic information, the fields of `S` vary depending on `OBJ`. For object specific details, refer to the help on the infoImpl method of that object.

comm.gpu.ViterbiDecoder.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `ACPR` 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)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.gpu.ViterbiDecoder.reset

Purpose Reset states of the GPU-based Viterbi Decoder modulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the GPU-based `ViterbiDecoder` object, `H`.

Purpose

Decode convolutionally encoded data using Viterbi algorithm

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

`Y = step(H,X)` decodes encoded data, `X`, using the Viterbi algorithm and returns `Y`. `X`, must be a column vector with data type and values that depend on how you set the `InputFormat` property. If the convolutional code uses an alphabet of 2^N possible symbols, the length of the input vector, `X`, must be `L*N` for some positive integer `L`. Similarly, if the decoded data uses an alphabet of 2^K possible output symbols, the length of the output vector, `Y`, is `L*K`.

`Y = step(H,X,R)` resets the internal states of the decoder when you input a non-zero reset signal, `R`. `R` must be a double precision, single precision or logical scalar. This syntax applies when you set the `TerminationMethod` property to `Continuous` and 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.

---

comm.HadamardCode

Purpose

Generate Hadamard code

Description

The `HadamardCode` object generates a Hadamard code from a Hadamard matrix, whose rows form an orthogonal set of codes. You can use orthogonal codes for spreading in communication systems in which the receiver is perfectly synchronized with the transmitter. In these systems, the despreading operation is ideal, because the codes decorrelate completely.

Construction

`H = comm.HadamardCode` creates a Hadamard code generator System object, `H`. This object generates Hadamard codes from a set of orthogonal codes.

`H = comm.HadamardCode(Name,Value)` creates a Hadamard code generator 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

Length

Length of generated code

Specify the length of the generated code as a numeric, integer scalar value with a power of two. The default is `64`.

Index

Row index of Hadamard matrix

Specify the row index of the Hadamard matrix as a numeric, integer scalar value in the range `[0, 1, ...  , N-1]`. N is the value of the `Length` property. The default is `60`. An $N \times N$ Hadamard matrix, denoted as $P(N)$, is defined recursively as follows: $P(1) = [1]$ $P(2N) = [P(N)\ P(N); P(N)\ -P(N)]$ The NxN Hadamard matrix has the property that $P(N) \times P(N)' = N \times$ `eye`(N). The `step` method outputs code samples from the row of the Hadamard matrix that you specify in this property.

When you set this property to an integer k, the output code has exactly k zero crossings, for $k = 0, 1, \ldots, N-1$.

SamplesPerFrame

Number of output samples per frame

Specify the number of Hadamard code samples that the `step` method outputs as a numeric, positive, integer scalar value. The default is `1`.

When you set this property to a value of M, the `step` method outputs M samples of a Hadamard code of length N. N equals the length of the code that you specify in the `Length` property.

OutputDataType

Data type of output

Specify the output data type as one of `double | int8`. The default is `double`.

Methods

clone	Create Hadamard code generator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of Hadamard code generator object
step	Generate Hadamard code

Examples

Generate 10 samples of a Hadamard code sequence with a length of 64.

```
hHCode = comm.HadamardCode('SamplesPerFrame', 10);
```

```
seq = step(hHCode)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Hadamard Code Generator block reference page. The object properties correspond to the block parameters, except:

- The object does not have a property to select frame based outputs.
- The object does not have a property that corresponds to the **Sample time** parameter.

See Also

`comm.WalshCode` | `comm.OVSFCode`

Purpose

Create Hadamard code generator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `HadamardCode` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.HadamardCode.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.HadamardCode.getNumOutputs

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.HadamardCode.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `HadamardCode` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.HadamardCode.reset

Purpose Reset states of Hadamard code generator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `HadamardCode` object, `H`.

Purpose

Generate Hadamard code

Syntax

```
Y = step(H)
```

Description

`Y = step(H)` outputs a frame of the Hadamard code in column vector `Y`. Specify the frame length with the `SamplesPerFrame` property. The Hadamard code corresponds to one of the rows of an *N*x*N* Hadamard matrix, where *N* is a nonnegative power of 2, which you specify in the `Length` property. Use the `Index` property to choose the row of the Hadamard matrix. The `step` method outputs the code in a bi-polar format with `0` and `1` mapped to `1` and `-1`, 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.

---

comm.HDLCRCDetector

Purpose

Detect errors in input data using HDL-optimized CRC

Description

This hardware-friendly cyclic redundancy code (CRC) detector System object computes checksums for its entire input frame. The `HDLCRCDetector` System object is optimized for HDL code generation. Instead of frame processing, the System object processes data at the streaming mode. Control signals are added at both input and output for easy data synchronization.

Construction

`H = comm.HDLCRCDetector` creates an HDL-optimized CRC detector System object, `H`, that detects errors in the input data according to a specified generator polynomial.

`H = comm.HDLCRCDetector(Name,Value,)` creates an HDL-optimized CRC detector System object, `H`, with additional options specified by one or more `Name,Value` pair arguments, where `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`H = comm.HDLCRCDetector(POLY,Name,Value)` creates an HDL-optimized CRC detector System object, `H`, with the `Polynomial` property set to POLY, and the other specified properties set to the specified values.

Input Arguments

POLY

Sets Polynomial property to POLY at System object construction

Properties

Polynomial

Specify the generator polynomial as a binary row vector, with coefficients in descending order of powers. If you set this property to a binary vector, its length must be equal to the degree of the polynomial plus 1. The default is [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1].

FinalXORValue

The value with which the CRC checksum is to be XORed just prior to being appended to the input data. This property can be specified as a binary, double or single precision data type scalar or vector. The vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. When you specify Final XOR Value as a scalar, the object expands the value to a row vector of length equal to the degree of the generator polynomial. The default is 0.

InitialState

Specify the initial conditions of the shift register as a binary, double or single precision data type scalar or vector. The vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. When you specify initial conditions as a scalar, the object expands the value to a row vector of length equal to the degree of the generator polynomial. The default is 0.

ReflectCRCChecksum

A logical quantity that specifies whether the output CRC checksum should be flipped around its center after the input data is completely through the shift register. The default is false.

ReflectInput

A logical quantity that specifies whether the input data should be flipped on a bytewise basis prior to entering the shift register. The default is false.

Methods

clone	Create HDLCRCDetector System object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics change

reset	Reset states of HDL CRC detector object
step	Generate CRC checksums for input message based on control signals and appends checksums to output message

Examples

Encode and decode a signal using an HDL-optimized CRC generator and detector.

1

Construct default polynomial with CRC length 16:

```
hGen = comm.HDLCRCGenerator;
hDet = comm.HDLCRCDetector;
```

2

Run `HDLCRCGenerator` 10 steps:

```
numSteps = 10;
```

3

Assign control signals for all 10 steps:

```
startIn = logical([1 0 0 0 0 0 0 0 0 0]);
endIn   = logical([0 1 0 0 0 0 0 0 0 0]);
validIn = logical([1 1 0 0 0 0 0 0 0 0]);
```

4

Assign 32 bit data to be encoded, in two 16 by 1 columns:

```
msg = randi([0 1],16,2);
```

5

Assign random input to the HDLCRCGenerator System object while it is processing msg:

```
randIn = randi([0, 1],16,numSteps-2);
dataIn = [msg randIn];
```

6

Run HDLCRCGenerator System object 10 steps:

```
% Output data: dataOutGen
% Output Control signals: startOutGen, endOutGen, validOutGen
for i =  1: numSteps
[dataOutGen(:,i),startOutGen(i),endOutGen(i),validOutGen(i)] = step(h
                           dataIn(:,i),startIn(i),endIn(i),validIn(i)
end
```

7

Add noise to encoded message:

```
dataOutGen(2,4) = ~dataOutGen(2,4);
```

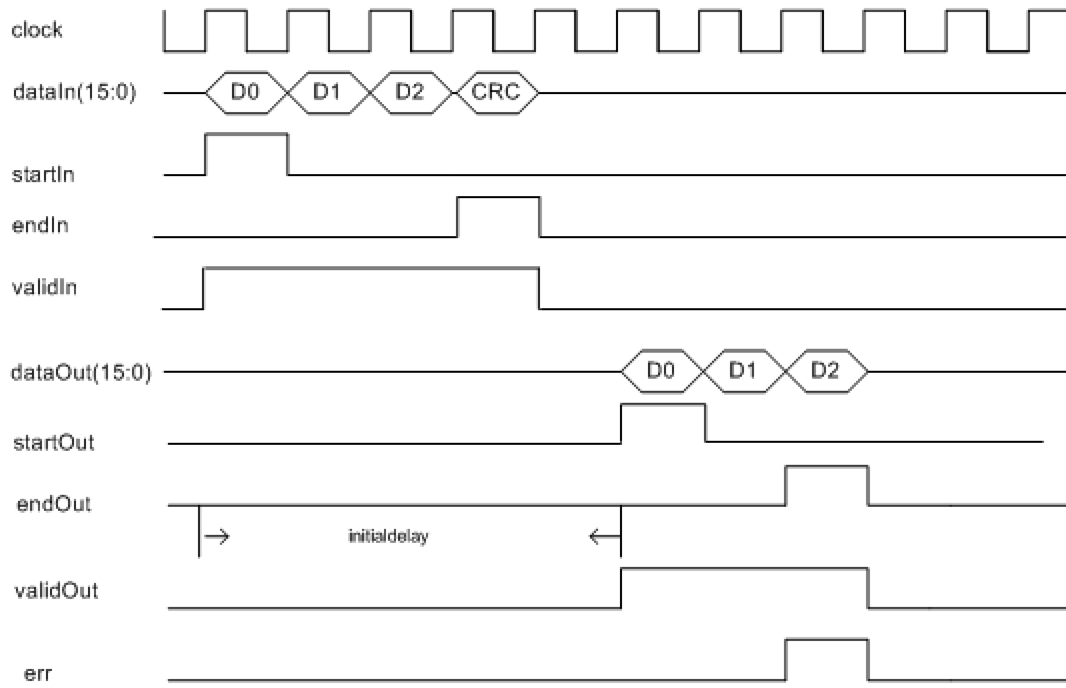
8

Run HDLCRCDetector System object 10 steps:

```
% Output data: dataOut
% Output Control signals: startOut, endOut, validOut,err
for i = 1:numSteps
[dataOut(:,i), startOut(i),endOut(i), validOut(i),err(i)] = step(hDet,
           dataOutGen(:,i),startOutGen(i),endOutGen(i),validOutGen(i)
end
```

Algorithms

Timing diagram for HDL-optimized CRC Detector



Initial Delay

The `HDLCRCGenerator` System object introduces a latency on the output. This latency can be computed with the following equation:

```
initialdelay =  3 * CRC length/input data width + 2
```

See Also

`comm.HDLCRCGenerator` | `comm.CRCDetector` |

Purpose

Create HDLCRCDetector System object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates another instance of the HDLCRCDetector System object, `H`, with the same property values. The clone method creates a new unlocked object with uninitialized states.

Input Arguments

H

HDL CRC Detector System object

Output Arguments

C

New instance of the HDLCRCDetector System object, `H`, with the same property values. The new unlocked object contains uninitialized states.

See Also

`comm.HDLCRCDetector` | `comm.HDLCRCDetector.isLocked` | `comm.HDLCRCDetector.release` | `comm.HDLCRCDetector.reset` | `comm.HDLCRCDetector.step` |

comm.HDLCRCDetector.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(H)`

Description

`L = isLocked(H)` returns the locked status, `L`, of the HDL CRC Detector 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.

Input Arguments

H

HDL CRC Detector System object

Output Arguments

L

Logical value. Either 1 (true) or 0 (false).

See Also

`comm.HDLCRCDetector` | `comm.HDLCRCDetector.clone` | `comm.HDLCRCDetector.release` | `comm.HDLCRCDetector.reset` | `comm.HDLCRCGenerator.step` |

Purpose

Allow property value and input characteristics change

Syntax

`release(H)`

Description

`release(H)` releases system resources (such as memory, file handles or hardware connections) of the HDL CRC Detector System object, `H`, and allows all its properties and input characteristics to be changed.

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.

Input Arguments

H

Instance of HDL CRC Detector System object

See Also

`comm.HDLCRCDetector` | `comm.HDLCRCDetector.clone` | `comm.HDLCRCDetector.isLocked` | `comm.HDLCRCDetector.reset` | `comm.HDLCRCDetector.step` |

comm.HDLCRCDetector.reset

Purpose Reset states of HDL CRC detector object

Syntax `reset(H)`

Description `reset(H)` resets the internal states of the HDL CRC Detector System object, `H`, to their initial values

Input Arguments

H

Instance of HDL CRC Detector System object

See Also `comm.HDLCRCDetector` | `comm.HDLCRCDetector.clone` | `comm.HDLCRCDetector.isLocked` | `comm.HDLCRCDetector.release` | `comm.HDLCRCDetector.step` |

Purpose

Generate CRC checksums for input message based on control signals and appends checksums to output message

Syntax

```
[Y,startOut,endOut,validOut,err] = step(H,X,startIn,endIn,
   validIn)
```

Description

`[Y,startOut,endOut,validOut,err] = step(H,X,startIn,endIn,validIn)` computes CRC checksums for an input message `X` based on the control signals and compares the computed checksum with input checksum. The output `err` is high if the two checksums are not equal.

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.

Input Arguments

H

`HDLCRCDetector` System object

X

Input message

- Must be a binary column vector.
- Data type can be double or logical.
- `X` can be part or all of the message to be encoded.
- The length of `X` should be less than or equal to the CRC length, and the CRC length should be divisible by the length of `X`.

- The CRC length is the order of the polynomial that you specify in the `Polynomial` property.

startIn

Indicates the start of an input message. `startIn` is scalar with logical data type.

endIn

Indicates the end of an input message. `endIn` is scalar with logical data type.

validIn

When `validIn` is high, input message is processed for CRC checksum computation. `validIn` is scalar with logical data type.

Output Arguments

Y

Object output, consisting of `X` + checksum plus the delay. Output `Y` has the same length and data type as input `X`.

startOut

Indicates the start of an output message. `startOut` is scalar with logical data type.

endOut

Indicates the end of an output message. `endOut` is scalar with logical data type.

validOut

Indicates the validation of output message plus CRC checksums.`validOut` is scalar with logical data type.

See Also

`comm.HDLCRCDetector` | `comm.HDLCRCDetector.clone` | `comm.HDLCRCDetector.isLocked` | `comm.HDLCRCDetector.release` | `comm.HDLCRCDetector.reset` |

Purpose

Generate HDL-optimized CRC code bits and append to input data

Description

This hardware-friendly CRC Generator System object, like the CRC Generator System object, generates cyclic redundancy code (CRC) bits. However, the HDL CRC Generator System object is optimized for HDL code generation. Instead of frame processing, the System object processes data at the streaming mode. Control signals are added at both input and output for easy data synchronization.

Construction

`H=comm.HDLCRCGenerator` creates an HDL-optimized cyclic redundancy code (CRC) generator System object, `H`. This object generates CRC bits according to a specified generator polynomial and appends them to the input data.

`H = comm.HDLCRCGenerator(Name,Value)` creates an HDL-optimized CRC generator System object, `H`, with additional options specified by one or more `Name,Value` pair arguments, where `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`H = comm.HDLCRCGenerator(POLY,Name,Value)` creates an HDL-optimized CRC generator System object, `H`, with the `Polynomial` property set to POLY, and the other specified properties set to the specified values.

Input Arguments

POLY

Sets Polynomial property to POLY at System object construction

Properties

Polynomial

Specify the generator polynomial as a binary row vector, with coefficients in descending order of powers. If you set this property to a binary vector, its length must be equal to the degree of the polynomial plus 1. The default is [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1].

FinalXORValue

The value with which the CRC checksum is to be XORed just prior to being appended to the input data. This property can be specified as a binary, double or single precision data type scalar or vector. The vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. When you specify Final XOR Value as a scalar, the object expands the value to a row vector of length equal to the degree of the generator polynomial. The default is 0.

InitialState

Specify the initial conditions of the shift register as a binary, double or single precision data type scalar or vector. The vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. When you specify initial conditions as a scalar, the object expands the value to a row vector of length equal to the degree of the generator polynomial. The default is 0.

ReflectCRCChecksum

A logical quantity that specifies whether the output CRC checksum should be flipped around its center after the input data is completely through the shift register. The default is false.

ReflectInput

A logical quantity that specifies whether the input data should be flipped on a bytewise basis prior to entering the shift register. The default is false.

Methods

clone	Create HDLCRCGenerator System object with same property values
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics change
reset	Reset states of CRC generator object
step	Generate CRC checksums for input message based on control signals and appends checksums to output message

Examples

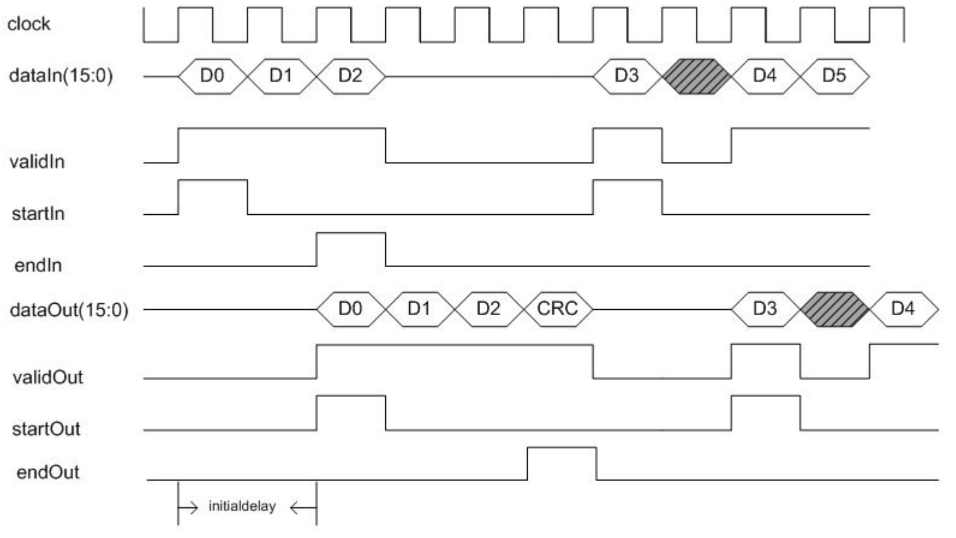
Encode signal using an HDL-optimized CRC generator.

```
% Using default polynomial with CRC length 16
hGen = comm.HDLCRCGenerator;

% run HDL CRC Generator 6 steps
numSteps = 6;

% Control signals for all 6 steps
startIn = logical([1 0 0 0 0 0]);
endIn   = logical([0 1 0 0 0 0]);
validIn = logical([1 1 0 0 0 0]);

% 32 bit data to be encoded, in two 16 by 1 columns
msg = randi([0 1],16,2);

% random input to HDLCRCGenerator while it is processing the msg
randIn = randi([0, 1],16,numSteps-2);
dataIn = [msg randIn];

% Run HDL CRC Generator 6 steps
% Output data: dataOut
% Output Control signals: startOut, endOut, validOut
for i =  1: numSteps
[dataOut(:,i), startOut(i),endOut(i), validOut(i)] = step(hGen,...
             dataIn(:,i),startIn(i),endIn(i),validIn(i));
end
```

Algorithms

Timing Diagram

Timing diagram for HDL-optimized CRC generator



Initial Delay

The HDL CRC Generator System object introduces a latency on the output. This latency can be computed with the following equation:

```
initialdelay = (CRC length/input data width) + 2
```

See Also

`comm.CRCGenerator` | `comm.HDLCRCGenerator` |

Purpose

Create HDLCRCGenerator System object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates another instance of the HDLCRCGenerator System object, `H`, with the same property values. The clone method creates a new unlocked object with uninitialized states.

Input Arguments

H

HDL CRC Generator System object

Output Arguments

C

New instance of the HDLCRCGenerator System object, `H`, with the same property values. The new unlocked object contains uninitialized states.

See Also

`comm.HDLCRCGenerator` | `comm.HDLCRCGenerator.isLocked` | `comm.HDLCRCGenerator.release` | `comm.HDLCRCGenerator.reset` | `comm.HDLCRCGenerator.step` |

comm.HDLCRCGenerator.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(H)`

Description

`L = isLocked(H)` returns the locked status, `L`, of the HDL CRC Generator 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.

Input Arguments

H

HDL CRC Generator System object

Output Arguments

L

Logical value. Either 1 (true) or 0 (false).

See Also

`comm.HDLCRCGenerator` | `comm.HDLCRCGenerator.clone` | `comm.HDLCRCGenerator.release` | `comm.HDLCRCGenerator.reset` | `comm.HDLCRCGenerator.step` |

Purpose

Allow property value and input characteristics change

Syntax

`release(H)`

Description

`release(H)` releases system resources (such as memory, file handles or hardware connections) of the HDL CRC Generator System object, `H`, and allows all its properties and input characteristics to be changed.

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.

Input Arguments

`H`

Instance of HDL CRC Generator System object

See Also

`comm.HDLCRCGenerator` | `comm.HDLCRCGenerator.clone` | `comm.HDLCRCGenerator.isLocked` | `comm.HDLCRCGenerator.reset` | `comm.HDLCRCGenerator.step` |

Purpose Reset states of CRC generator object

Syntax

```
reset(H)
```

Description `reset(H)` resets the internal states of the HDL CRC Generator System object, `H`, to their initial values

Input Arguments

H

Instance of HDL CRC Generator System object

See Also `comm.HDLCRCGenerator` | `comm.HDLCRCGenerator.clone` | `comm.HDLCRCGenerator.isLocked` | `comm.HDLCRCGenerator.release` | `comm.HDLCRCGenerator.step` |

Purpose

Generate CRC checksums for input message based on control signals and appends checksums to output message

Syntax

```
[Y,startOut,endOut,validOut] = step(H,X,startIn,endIn,
   validIn)
```

Description

`[Y,startOut,endOut,validOut] = step(H,X,startIn,endIn, validIn)` generates CRC checksums for input message `X` based on control signals and appends the checksums to `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.

Input Arguments

H

HDL CRC Generator System object

X

Input message

- Must be a binary column vector.
- Data type can be double or logical.
- `X` can be part or all of the message to be encoded.
- The length of `X` should be less than or equal to the CRC length, and the CRC length should be divisible by the length of `X`.
- The CRC length is the order of the polynomial that you specify in the `Polynomial` property.

startIn

Indicates the start of an input message. `startIn` is scalar with logical data type.

endIn

Indicates the end of an input message. `endIn` is scalar with logical data type.

validIn

When `validIn` is high, input message is processed for CRC checksum computation. `validIn` is scalar with logical data type.

Output Arguments

Y

Object output, consisting of `X` + checksum plus the delay. Output `Y` has the same length and data type as input `X`.

startOut

Indicates the start of an output message. `startOut` is scalar with logical data type.

endOut

Indicates the end of an output message. `endOut` is scalar with logical data type.

validOut

Indicates the validation of output message plus CRC checksums.`validOut` is scalar with logical data type.

See Also

`comm.HDLCRCGenerator` | `comm.HDLCRCGenerator.clone` | `comm.HDLCRCGenerator.isLocked` | `comm.HDLCRCGenerator.release` | `comm.HDLCRCGenerator.reset` |

Purpose

Decode data using a Reed-Solomon decoder

Description

The HDL-optimized `HDLRSDecoder` System object recovers a message vector from a Reed-Solomon codeword vector. For proper decoding, the property values for this object should match those in the corresponding `HDLRSEncoder` System object.

Construction

`H = comm.HDLRSDecoder` creates an HDL-optimized RS decoder System object, `H`, that performs Reed-Solomon (RS) decoding.

`H = comm.HDLRSDecoder(Name,Value)` creates an HDL-optimized RS decoder System object, `H`, with additional options specified by one or more `Name,Value` pair arguments, where Name is a property name and Value is the corresponding value. Name must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`H = comm.HDLRSDecoder(N,K,Name,Value)` creates an HDL-optimized RS decoder System object, `H`, with the CodewordLength property set to N, the MessageLength property set to K, and other specified property Names set to the specified Values.

Properties

B

B value for polynomial generation

BSource

Source of B, the starting power for roots of the primitive polynomial

Specify the source of the B value as one of these values:

- Auto: B=0
- Property

Default: Auto

CodewordLength

Codeword length

Specify the codeword length of the RS code as a double-precision, positive, integer scalar value. The default is 7.

If you set the PrimitivePolynomialSource property to Auto, CodewordLength must be in the range 3 < CodewordLength ≤ 2^{16}–1.

When you set the PrimitivePolynomialSource property to Property, CodewordLength must be in the range 3 ≤ CodewordLength ≤ 2^M–1. M is the degree of the primitive polynomial that you specify with the PrimitivePolynomialSource and PrimitivePolynomial properties. M must be in the range 3≤ M ≤ 16. The difference (CodewordLength –MessageLength) must be an even integer. The value of this property is rounded up to 2^M–1.

If the value of this property is less than 2^M–1, the object assumes a shortened RS code.

MessageLength

Message length

Specify the message length as a double-precision, positive integer scalar value. The default is 3. The difference (CodewordLength – MessageLength) must be an even integer.

NumErrorsOutputPort

Enable number of errors output

When you set this property to true, the step method outputs number of corrected errors. The number of corrected errors is not valid when errOut is asserted, since there were more errors than could be corrected. The default is false.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as `Auto | Property`. The default is `Auto`.

When you set this property to `Auto`, the object uses a primitive polynomial of degree M = `ceil(`**log2**`(CodewordLength+1))`, which is the result of `fliplr(de2bi(primpoly(`M`)))`.

When you set this property to `Property`, you can specify a polynomial using the `PrimitivePolynomial` property.

PrimitivePolynomial

Primitive polynomial

Specify the primitive polynomial that defines the finite field `GF`(2^M) corresponding to the integers that form messages and codewords. You must set this property to a double-precision, binary row vector that represents a primitive polynomial over `GF`(2) of degree M in descending order of powers.

This property applies when you set the `PrimitivePolynomialSource` property to `Property`.

Methods

clone	Create HDLRSDecoder System object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics change
step	Perform Reed-Solomon decoding

Examples

RS-encode and decode a DVD-II standard packet of random data.

1

Assign data and create System objects.

```
hHDLEnc = comm.HDLRSEncoder(204,188,'BSource','Property','B',0);
```

```
hHDLDec = comm.HDLRSDecoder(204,188,'BSource','Property','B',0);
dataIn = [randi([0,255],188,1,'uint8') ; zeros(1024-188,1)];
for ii = 1:1024
  [encOut(ii), startOut(ii), endOut(ii), validOut(ii)] = step(hHDLEnc, da
  [decOut(ii), decStartOut(ii), decEndOut(ii), decValidOut(ii), decErrOut
end
```

2

Check results.

```
assert(all(dataIn(1:188) == decOut(decValidOut)'))
```

See Also

comm.RSDecoder | comm.HDLRSEncoder

Purpose

Create HDLRSDecoder System object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates another instance of the HDLRSDecoder System object, `H`, with the same property values. The clone method creates a new unlocked object with uninitialized states.

Input Arguments

H

HDLRSDecoder System object

Output Arguments

C

New instance of the HDLRSDecoder System object, `H`, with the same property values. The new unlocked object contains uninitialized states.

See Also

`comm.HDLRSDecoder` | `comm.HDLRSDecoder.isLocked` | `comm.HDLRSDecoder.release` | `comm.HDLRSDecoder.step` |

comm.HDLRSDecoder.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
L = isLocked(H)
```

Description

`L = isLocked(H)` returns the locked status, `L`, of the HDLRSDecoder 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.

Input Arguments

H

HDLRSDecoder System object

Output Arguments

L

Logical value. Either 1 (true) or 0 (false).

See Also

`comm.HDLRSDecoder` | `comm.HDLRSDecoder.clone` | `comm.HDLRSDecoder.release` | `comm.HDLRSDecoder.step` |

comm.HDLRSDecoder.release

Purpose

Allow property value and input characteristics change

Syntax

`release(H)`

Description

`release(H)` releases system resources (such as memory, file handles or hardware connections) of the HDLRSDecoder System object, `H`, and allows all its properties and input characteristics to be changed.

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.

Input Arguments

H

Instance of HDLRSDecoder System object

See Also

`comm.HDLRSDecoder` | `comm.HDLRSDecoder.clone` | `comm.HDLRSDecoder.isLocked` | `comm.HDLRSDecoder.step` |

Purpose

Perform Reed-Solomon decoding

Syntax

```
[Y,startOut,endOut,validOut,errOut] = step(H,X,startIn,EndIn,
   validIn)
```

Description

`[Y,startOut,endOut,validOut,errOut] = step(H,X,startIn,EndIn,validIn)` decodes the input data, `X`, and returns the encoded data, `Y`, of `HDLRSDecoder` System object, `H`.

The step method for this object accepts fixed-point (fi) inputs for `X`.

Note Calling step on an object puts that object into a locked state. When locked, you cannot change nontunable properties or any input characteristics (size, data type and complexity) without reinitializing (unlocking and relocking) the object.

Input Arguments

H

Instance of `HDLRSDecoder` System object

X

Message data

Must be an integer (uint8, uint16, uint32) or fi(). Doubles are allowed for simulation but not for HDL code generation.

startIn

Indicates the start of a frame of data. Boolean value.

endIn

Indicates the end of a frame of data. Boolean value.

validIn

Indicates that input data is valid. Boolean value.

Output Arguments

Y

Message data followed by parity words

The data type is the same as the input data port. Will be an integer (uint8, uint16, uint32) or fi(). Doubles are allowed for simulation but not for HDL code generation.

startOut

Indicates the start of a frame of data. Boolean value.

endOut

Indicates the end of a frame of data, including checksum. Boolean value.

validOut

Indicates that output data is valid. Boolean value.

errOut

Indicates the corruption of the received data when error is high. Boolean in and out.

Examples

RS-encode and decode a DVD-II standard packet of random data.

1

Assign data and create System objects.

```
hHDLEnc = comm.HDLRSEncoder(204,188,'BSource','Property','B',0);
hHDLDec = comm.HDLRSDecoder(204,188,'BSource','Property','B',0);
dataIn = [randi([0,255],188,1,'uint8') ; zeros(1024-188,1)];
for ii = 1:1024
  [encOut(ii), startOut(ii), endOut(ii), validOut(ii)] = step(hHDLEnc,
  [decOut(ii), decStartOut(ii), decEndOut(ii), decValidOut(ii), decErr
end
```

2

Check results.

```
assert(all(dataIn(1:188) == decOut(decValidOut)'))
```

See Also

comm.HDLRSDecoder | comm.HDLRSDecoder.clone |
comm.HDLRSDecoder.isLocked | comm.HDLRSDecoder.release |

Purpose

Encode data using a Reed-Solomon encoder

Description

The HDL-optimized `HDLRSEncoder` System object creates a Reed-Solomon code with message and codeword lengths you specify.

Construction

`H = comm.HDLRSEncoder` returns a block encoder System object, `H`, that performs Reed-Solomon (RS) encoding in a streaming fashion for HDL.

`H = comm.HDLRSEncoder(Name,Value,)` creates an HDL-optimized block encoder System object, `H`, with additional options specified by one or more `Name,Value` pair arguments, where `Name` is a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`H = comm.HDLRSEncoder(N,K,Name,Value)` creates an RS encoder object, `H`, with the CodewordLength property set to `N`, the MessageLength property set to `K`, and other specified property `Name, Value` pair arguments.

Properties

B

B value for polynomial generation

BSource

Source of B, the starting power for roots of the primitive polynomial

Specify the source of the B value as one of these values:

- Auto: B=0
- Property

Default: Auto

CodewordLength

Codeword length

Specify the codeword length of the RS code as a double-precision, positive, integer scalar value. The default is `7`.

If you set the `PrimitivePolynomialSource` property to `Auto`, `CodewordLength` must be in the range 3 < `CodewordLength` ≤ 2^{16}–1.

When you set the `PrimitivePolynomialSource` property to `Property`, `CodewordLength` must be in the range 3 ≤ `CodewordLength` ≤ 2^M–1. M is the degree of the primitive polynomial that you specify with the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties. M must be in the range 3≤ M ≤ 16. The difference (`CodewordLength` –`MessageLength`) must be an even integer. The value of this property is rounded up to 2^M–1.

If the value of this property is less than 2^M–1, the object assumes a shortened RS code.

MessageLength

Message length

Specify the message length as a double-precision, positive integer scalar value. The default is `3`. The difference (`CodewordLength` – `MessageLength`) must be an even integer.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as `Auto | Property`. The default is `Auto`.

When you set this property to `Auto`, the object uses a primitive polynomial of degree M = `ceil`(**log2**(`CodewordLength+1`)), which is the result of `fliplr`(`de2bi`(`primpoly`(M))).

When you set this property to `Property`, you can specify a polynomial using the `PrimitivePolynomial` property.

PrimitivePolynomial

Primitive polynomial

Specify the primitive polynomial that defines the finite field `GF`(2^M) corresponding to the integers that form messages and codewords. You must set this property to a double-precision, binary row vector that represents a primitive polynomial over `GF`(2) of degree M in descending order of powers.

This property applies when you set the `PrimitivePolynomialSource` property to `Property`.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as `None | Property`. The default is `None`. If you set this property to `None` then the object does not apply puncturing to the code. If you set this property to `Property` then the object punctures the code based on a puncture pattern vector specified in the `PuncturePattern` property.

PuncturePattern

Puncture pattern vector

Specify the pattern used to puncture the encoded data as a double-precision, binary column vector with a length of (`CodewordLength`–`MessageLength`). The default is `[ones(2,1); zeros(2,1)]`. Zeros in the puncture pattern vector indicate the position of the parity symbols that are punctured or excluded from each codeword. This property applies when you set the `PuncturePatternSource` property to `Property`.

Methods

clone	Create HDLRSEncoder System object with same property values
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics change
step	Perform Reed-Solomon encoding

Examples

RS-encode a DVD-II standard packet of random data.

```
hHDLEnc = comm.HDLRSEncoder(204,188,'BSource','Property','B',0);
hRSEnc  = comm.RSEncoder(204,188,...
                             'GeneratorPolynomialSource','Property',...
                             'GeneratorPolynomial',rsgenpoly(255,239,[],0
dataIn = [randi([0,255],188,1,'uint8') ; zeros(255-188,1)];
for ii = 1:255
  [dataOut(ii), startOut(ii), endOut(ii), validOut(ii)] = step(hHDLEn
end
% Check the result:
Y = step(hRSEnc,dataIn(1:188));
assert(all(Y == dataOut(validOut)'))
```

See Also

comm.RSEncoder | comm.HDLRSDecoder

Purpose

Create HDLRSEncoder System object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates another instance of the HDLRSEncoder System object, `H`, with the same property values. The clone method creates a new unlocked object with uninitialized states.

Input Arguments

`H`

HDLRSEncoder System object

Output Arguments

`C`

New instance of the HDLRSEncoder System object, `H`, with the same property values. The new unlocked object contains uninitialized states.

See Also

`comm.HDLRSEncoder` | `comm.HDLRSEncoder.isLocked` | `comm.HDLRSEncoder.release` | `comm.HDLRSEncoder.step` |

comm.HDLRSEncoder.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`L = isLocked(H)`

Description

`L = isLocked(H)` returns the locked status, `L`, of the HDLRSEncoder 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.

Input Arguments

H

HDLRSEncoder System object

Output Arguments

L

Logical value. Either 1 (true) or 0 (false).

See Also

`comm.HDLRSEncoder` **|** `comm.HDLRSEncoder.clone` **|** `comm.HDLRSEncoder.release` **|** `comm.HDLRSEncoder.step` **|**

Purpose

Allow property value and input characteristics change

Syntax

`release(H)`

Description

`release(H)` releases system resources (such as memory, file handles or hardware connections) of the HDLRSEncoder System object, `H`, and allows all its properties and input characteristics to be changed.

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.

Input Arguments

`H`

Instance of HDLRSEncoder System object

See Also

`comm.HDLRSEncoder` | `comm.HDLRSEncoder.clone` | `comm.HDLRSEncoder.isLocked` | `comm.HDLRSEncoder.step` |

Purpose

Perform Reed-Solomon encoding

Syntax

```
[Y,startOut,endOut,validOut] = step(H,X,startIn,EndIn,
   validIn)
```

Description

`[Y,startOut,endOut,validOut] = step(H,X,startIn,EndIn,validIn)` decodes the input data, `X`, and returns the encoded data, `Y`, of `HDLRSEncoder` System object, `H`.

The step method for this object accepts fixed-point (fi) inputs for `X`.

Note Calling step on an object puts that object into a locked state. When locked, you cannot change nontunable properties or any input characteristics (size, data type and complexity) without reinitializing (unlocking and relocking) the object.

Input Arguments

H

Instance of `HDLRSEncoder` System object

X

Message data

Must be an integer (uint8, uint16, uint32) or fi(). Doubles are allowed for simulation but not for HDL code generation.

startIn

Indicates the start of a frame of data. Boolean value.

endIn

Indicates the end of a frame of data. Boolean value.

validIn

Indicates that input data is valid. Boolean value.

Output Arguments

Y

Message data followed by parity words

The data type is the same as the input data port. Will be an integer (uint8, uint16, uint32) or fi(). Doubles are allowed for simulation but not for HDL code generation.

startOut

Indicates the start of a frame of data. Boolean value.

endOut

Indicates the end of a frame of data, including checksum. Boolean value.

validOut

Indicates that output data is valid. Boolean value.

Examples

RS-encode and decode a DVD-II standard packet of random data.

1

Assign data and create System objects.

```
hHDLEnc = comm.HDLRSEncoder(204,188,'BSource','Property','B',0);
hHDLDec = comm.HDLRSDecoder(204,188,'BSource','Property','B',0);
dataIn = [randi([0,255],188,1,'uint8') ; zeros(1024-188,1)];
for ii = 1:1024
  [encOut(ii), startOut(ii), endOut(ii), validOut(ii)] = step(hHDLEnc,
  [decOut(ii), decStartOut(ii), decEndOut(ii), decValidOut(ii), decErr
end
```

2

Check results.

```
assert(all(dataIn(1:188) == decOut(decValidOut)'))
```

See Also

comm.HDLRSEncoder | comm.HDLRSEncoder.clone | comm.HDLRSEncoder.isLocked | comm.HDLRSEncoder.release |

comm.HelicalDeinterleaver

Purpose

Restore ordering of symbols using helical array

Description

The `HelicalDeinterleaver` object permutes the symbols in the input signal by placing them in a row-by-row array and then selecting groups helically to send to the output port.

Construction

`H = comm.HelicalDeinterleaver` creates a helical deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using the helical interleaver System object.

`H = comm.HelicalDeinterleaver(Name,Value)` creates a helical deinterleaver 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

NumColumns

Number of columns in helical array

Specify the number of columns in the helical array as a positive integer scalar value. The default is `6`.

GroupSize

Size of each group of input symbols

Specify the size of each group of input symbols as a positive integer scalar value. The default is `4`.

StepSize

Helical array step size

Specify number of rows of separation between consecutive input groups in their respective columns of the helical array. This property requires a positive integer scalar value. The default is `1`.

InitialConditions

Initial conditions of helical array

Specify the value that is initially stored in the helical array as a numeric scalar value. The default is `0`.

Methods

clone	Create helical deinterleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the helical deinterleaver object
step	Restore ordering of symbols using a helical array

Examples

Interleave and deinterleave random data.

```
hInt = comm.HelicalInterleaver('GroupSize', 2, ...
                'NumColumns', 3, ...
                'InitialConditions', -1);
hDeInt = comm.HelicalDeinterleaver('GroupSize', 2, ...
                'NumColumns', 3, ...
                'InitialConditions', -1);
data = randi(7, 6, 1);
intData = step(hInt, data);
deIntData = step(hDeInt, intData);

% compare the original sequence, interleaved sequence, and restore
[data, intData, deIntData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Helical Deinterleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.HelicalInterleaver` | `comm.MultiplexedDeinterleaver`

Purpose

Create helical deinterleaver object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `HelicalDeinterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.HelicalDeinterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.HelicalDeinterleaver.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `HelicalDeinterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.HelicalDeinterleaver.reset

Purpose Reset states of the helical deinterleaver object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `HelicalDeinterleaver` object, `H`.

Purpose Restore ordering of symbols using a helical array

Syntax

```
Y = step(H,X)
```

Description `Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a helical interleaver and returns `Y`. The input `X` must be a column vector. The data type must be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The helical deinterleaver object uses an array for its computations. If you set the `NumColumns` property of the object to C, then the array has C columns and unlimited rows. If you set the `GroupSize` property to N, then the object accepts an input of length $C \times N$ and inserts it into the next N rows of the array. The object also places the value of the `InitialConditions` property into certain positions in the top few rows of the array. This accommodates the helical pattern and also preserves the vector indices of symbols that pass through the `HelicalInterleaver` and `HelicalDeinterleaver` objects. The output consists of consecutive groups of N symbols. The object selects the k-th output group in the array from column k mod C. This selection is of type helical because of the reduction modulo C and because the first symbol in the k-th group is in row $1+(k\text{-}1) \times s$, where s is the value for the `StepSize` 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

Permute input symbols using helical array

Description

The `HelicalInterleaver` object permutes the symbols in the input signal by placing them in an array in a helical arrangement and then sending rows of the array to the output port.

Construction

`H = comm.HelicalInterleaver` creates a helical interleaver System object, `H`. This object permutes the input symbols in the input signal by placing them in an array in a helical arrangement.

`H = comm.HelicalInterleaver(Name,Value)` creates a helical interleaver 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

NumColumns

Number of columns in helical array

Specify the number of columns in the helical array as a positive integer scalar value. The default is `6`.

GroupSize

Size of each group of input symbols

Specify the size of each group of input symbols as a positive integer scalar value. The default is `4`.

StepSize

Helical array step size

Specify the number of rows of separation between consecutive input groups in their respective columns of the helical array. This property requires as a positive integer scalar value . The default is `1`.

InitialConditions

Initial conditions of helical array

Specify the value that is initially stored in the helical array as a numeric scalar value. The default is `0`.

Methods

clone	Create helical interleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the helical interleaver object
step	Permute input symbols using a helical array

Examples

Interleave and deinterleave random data.

```
hInt = comm.HelicalInterleaver('GroupSize', 2, ...
                'NumColumns', 3, ...
                'InitialConditions', -1);
hDeInt = comm.HelicalDeinterleaver('GroupSize', 2, ...
                'NumColumns', 3, ...
                'InitialConditions', -1);
data = randi(7, 6, 1);
intData = step(hInt, data);
deIntData = step(hDeInt, intData);

% compare the original sequence, interleaved sequence, and restore
[data, intData, deIntData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Helical Interleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.HelicalDeinterleaver` | `comm.MultiplexedInterleaver`

Purpose

Create helical interleaver object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `HelicalInterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.HelicalInterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.HelicalInterleaver.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `HelicalInterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.HelicalInterleaver.reset

Purpose Reset states of the helical interleaver object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `HelicalInterleaver` object, `H`.

Purpose

Permute input symbols using a helical array

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The input `X` must be a column vector. The data type must be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The helical interleaver object places the elements of `X` in an array in a helical fashion. If you set the `NumColumns` property of the object to C, then the array has C columns and unlimited rows. If you set the `GroupSize` property to N, then the object accepts an input of length $C\times N$ and partitions the input into consecutive groups of N symbols. The object places the k-th group in the array along column k mod C. This placement is of type helical because of the reduction modulo C and because the first symbol in the k-th group is in the row $1+(k\text{-}1)\times s$, where s is the value for the `StepSize` property. Positions in the array that do not contain input symbols have default contents specified by the `InitialConditions` property. The object outputs $C\times N$ symbols from the array by reading the next N rows sequentially.

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 vector of integers to vector of bits

Description

The `IntegerToBit` object maps each integer (or fixed-point value) in the input vector to a group of bits in the output vector.

Construction

`H = comm.IntegerToBit` creates an integer-to-bit converter System object, `H`. This object maps a vector of integer-valued or fixed-point inputs to a vector of bits.

`H = comm.IntegerToBit(Name,Value)` creates an integer-to-bit converter 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`).

`H = comm.IntegerToBit(NUMBITS,Name,Value)` creates an integer-to-bit converter object, `H`. This object has the `BitsPerInteger` property set to `NUMBITS` and the other specified properties set to the specified values.

Properties

BitsPerInteger

Number of bits per integer

Specify the number of bits the System object uses to represent each input integer. You must set this property to a scalar integer between `1` and `32`. The default is `3`.

MSBFirst

Output bit words with first bit as most significant bit

Set this property to true to indicate that the first bit of the output bit words is the most significant bit (MSB). The default is `true`. Set this property to `false` to indicate that the first bit of the output bit words is the least significant bit (LSB).

SignedIntegerInput

Assume inputs are signed integers

Set this property to true if the integer inputs are signed. The default is false. Set this property to false if the integer inputs are unsigned. If the SignedIntegerInput property is false, the input values must be between 0 and (2^N)–1. In this case, N is the value you specified in the BitsPerInteger property. When you set this property to true, the input values must be between $-(2^{(N-1)})$ and $(2^{(N-1)})-1$.

OutputDataType

Data type of output

Specify output data type as one of Full precision | Smallest unsigned integer | Same as input | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | logical. The default is Full precision.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in Smallest unsigned integer or Full precision mode.

When you set this property to Full precision, the object determines the output data type based on the input data type. If the input data type is double- or single-precision, the output data has the same data type as the input data. Otherwise, the output data type is determined in the same way as when you set this property to Smallest unsigned integer.

When you set this property to Same as input, and the input data type is numeric or fixed-point integer (fi object), the output data has the same data type as the input data.

Methods

clone	Create an integer-to-bit converter object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Convert vector of integers to vector of bits

Examples

Convert randomly generated integers to 4-bit words.

```
hIntToBit = comm.IntegerToBit(4);
intData = randi([0 2^hIntToBit.BitsPerInteger-1],3,1);
bitData = step(hIntToBit,intData);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Integer to Bit Converter block reference page. The object properties correspond to the block parameters.

See Also

comm.BitToInteger | de2bi | dec2bin

Purpose

Create an integer-to-bit converter object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `IntegerToBit` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.IntegerToBit.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.IntegerToBit.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `IntegerToBit` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 vector of integers to vector of bits

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` converts integer input, `X`, to corresponding bits, `Y`. The input must be scalar or a column vector and the data type can be numeric or fixed-point (fi objects). The output is a column vector with length equal to length(`X`)$\times N$, where N is the value of the `BitsPerInteger` property. If any input value is outside the range of N, the object issues an error. If the `SignedIntegerInput` property is `false`, the input values must be between `0` and (2^N)-1. If you set the `SignedIntegerInput` property to `true`, the input values must be between $-(2^{(N-1)})$ and $(2^{(N-1)})$-1.

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

Integrate discrete-time signal with periodic resets

Description

The `IntegrateAndDumpFilter` object creates a cumulative sum of the discrete-time input signal, while resetting the sum to zero according to a fixed schedule. When the simulation begins, the object discards the number of samples specified in the Offset property. After this initial period, the object sums the input signal along columns and resets the sum to zero every *N*input samples, set by the integration period property. The reset occurs after the object produces output at that time step.

Construction

`H = comm.IntegrateAndDumpFilter` creates an integrate and dump filter System object, `H`. this object integrates over a number of samples in an integration period, and then resets at the end of that period.

`H = comm.IntegrateAndDumpFilter(Name,Value)` creates an integrate and dump filter 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`).

`H = comm.IntegrateAndDumpFilter(PERIOD,Name,Value)` creates an integrate and dump filter object, `H`. This object has the `IntegrationPeriod` property set to `PERIOD` and the other specified properties set to the specified values.

Properties

IntegrationPeriod

Integration period

Specify the integration period, in samples, as a positive, integer scalar value greater than `1`. The integration period defines the length of the sample blocks that the object integrates between resets. The default is `8`.

Offset

Number of offset samples

Specify a nonnegative, integer vector or scalar specifying the number of input samples that the object discards from each column of input data at the beginning of data processing. Discarding begins when you call the step method for the first time. The default is 0.

When you set the Offset property to a nonzero value, the object outputs one or more zeros during the initial period while discarding input samples.

When you specify this property as a vector of length L, the i-th element of the vector corresponds to the offset for the i-th column of the input data matrix, which has L columns.

When you specify this property as a scalar value, the object applies the same offset to each column of the input data matrix. The offset creates a transient effect, rather than a persistent delay.

DecimateOutput

Decimate output

Specify whether the step method returns intermediate cumulative sum results or decimates intermediate results. The default is true.

When you set this property to true, the step method returns one output sample, consisting of the final integration value, for each block of IntegrationPeriod input samples. If the inputs are ($K\times$ IntegrationPeriod)$\times L$ matrices, then the outputs are $K\times L$ matrices.

When you set this property to false, the step method returns IntegrationPeriod output samples, comprising the intermediate cumulative sum values, for each block of IntegrationPeriod input samples. In this case, inputs and outputs have the same dimensions.

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 of fixed-point numeric values

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 when fixed-point numeric values overflow

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.

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as one of Full precision | Same as input | Custom. The default is Full precision. When you set this property to Full precision the object automatically calculates the accumulator output word and fraction lengths. Set this property to Custom to specify the accumulator data type using the CustomAccumulatorDataType property. This property applies when you set the FullPrecisionOverride property to false.

CustomAccumulatorDataType

Fixed-point data type of accumulator

Specify the accumulator fixed-point type as a scaled `numerictype` object with a signedness of `Auto`. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` property to false and the `AccumulatorDataType` property to `Custom`.

OutputDataType

Data type of output

Specify the output fixed-point type as one of `Same as accumulator | Same as input | Custom`. The default is `Same as accumulator`. This property applies when you set the `FullPrecisionOverride` property to false.

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a scaled `numerictype` object with a signedness of `Auto`. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` property to false and the `OutputDataType` property to `Custom`.

Methods

clone	Create integrate and dump filter object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Integrate discrete-time signal with periodic resets

Examples

Integrate a signal specifying an integration period of 5 samples.

```
hInt = comm.IntegrateAndDumpFilter(5);
hInt.Offset = 3;
% Data matrix contains three columns (i.e. three channels)
data = reshape(1:30, 10, 3);
result = step(hInt, data)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Integrate and Dump block reference page. The object properties correspond to the block parameters, except:

The **Output intermediate values** parameter corresponds to the `DecimateOutput` property.

comm.IntegrateAndDumpFilter.clone

Purpose Create integrate and dump filter object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `IntegrateAndDumpFilter` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.IntegrateAndDumpFilter.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `IntegrateAndDumpFilter` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Integrate discrete-time signal with periodic resets

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` periodically integrates blocks of N samples from the input data, `X`, and returns the result in `Y`. N is the number of samples that you specify in the `IntegrationPeriod` property. `X` is a column vector or a matrix and the data type is double, single or fixed-point (fi objects). `X` must have K*N rows for some positive integer K, with one or more columns. The object treats each column as an independent channel with integration occurring along every column. The dimensions of output `Y` depend on the value you set for the `DecimateOutput` 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.

---

comm.KasamiSequence

Purpose

Generate Kasami sequence

Description

The `KasamiSequence` object generates a sequence from the set of Kasami sequences. The Kasami sequences are a set of sequences that have good cross-correlation properties.

Construction

`H = comm.KasamiSequence` creates a KasamiSequence System object, `H`. This object generates a Kasami sequence.

`H = comm.KasamiSequence(Name,Value)` creates a Kasami sequence generator 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

Polynomial

Generator polynomial

Specify the polynomial that determines the shift register's feedback connections. The default is `[1 0 0 0 0 1 1]`.

You can specify the generator polynomial as a binary numeric vector that lists the coefficients of the polynomial in descending order of powers. The first and last elements must equal `1`. Specify the length of this vector as n+1, where n is the degree of the generator polynomial and must be even.

Alternatively, you can specify the generator polynomial as a vector containing the exponents of z for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`. For example, `[1 0 0 0 0 0 1 0 1]` and `[8 2 0]` represent the same polynomial, $g(z) = z^8 + z^2 + 1$.

InitialConditions

Initial conditions of shift register

Specify the initial values of the shift register as a binary numeric scalar or as binary numeric vector. The default is `[0 0 0 0 0`

1]. Set the vector length equal to the degree of the generator polynomial.

When you set this property to a vector value, each element of the vector corresponds to the initial value of the corresponding cell in the shift register.

When you set this property to a scalar value, that value specifies the initial conditions of all the cells of the shift register. The scalar, or at least one element of the specified vector, requires a nonzero value for the object to generate a nonzero sequence.

Index

Sequence index

Specify the index to select a Kasami sequence of interest from the set of possible sequences. The default is 0. Kasami sequences have a period equal to $N = 2^n - 1$, where n indicates a nonnegative, even integer equal to the degree of the generator polynomial that you specify in the `Polynomial` property.

There are two classes of Kasami sequences: those obtained from a small set and those obtained from a large set. You choose a Kasami sequence from the small set by setting this property to a numeric, scalar, integer value in the range $[0 \ldots 2^{n/2}–2]$. You choose a sequence from the large set by setting this property to a numeric 1×2 integer vector $[k\ m]$ for k in $[–2, \ldots, 2^n–2]$, and m in $[–1, \ldots, 2^{n/2}–2]$.

Shift

Sequence offset from initial time

Specify the offset of the Kasami sequence from its starting point as a numeric, integer scalar value that can be positive or negative. The default is 0. The Kasami sequence has a period of $N = 2^n–1$, where n is the degree of the generator polynomial that you specify in the `Polynomial` property. The shift value is wrapped with respect to the sequence period.

VariableSizeOutput

Enable variable-size outputs

Set this property to true to enable an additional input to the step method. The default is false. When you set this property to true, the enabled input specifies the output size of the Kasami sequence used for the step. The input value must be less than or equal to the value of the `MaximumOutputSize` property.

When you set this property to false, the `SamplesPerFrame` property specifies the number of output samples.

MaximumOutputSize

Maximum output size

Specify the maximum output size of the Kasami sequence as a positive integer 2-element row vector. The second element of the vector must be 1. The default is [10 1].

This property applies when you set the `VariableSizeOutput` property to true.

SamplesPerFrame

Number of output samples per frame

Specify the number of Kasami sequence samples that the step method outputs as a numeric, positive, integer scalar value . The default value is `1`.

When you set this property to a value of M, then the `step` method outputs M samples of a Kasami sequence that has a period of $N = 2^n-1$. The value n equals the degree of the generator polynomial that you specify in the `Polynomial` property.

ResetInputPort

Enable generator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. The additional input resets the states of the Kasami sequence generator to the initial conditions that you specify in the `InitialConditions` property.

OutputDataType

Data type of output

Specify the output data type as one of `double | logical`. The default is `double`.

Methods

clone	Create Kasami sequence generator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of Kasami sequence generator object
step	Generate a Kasami sequence

Examples

Generate 5 samples of a Kasami sequence of length 63.

```
hks = comm.KasamiSequence('SamplesPerFrame', 5);
x   = step(hks)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Kasami Sequence Generator block reference page. The object properties correspond to the block parameters, except:

- The object does not have a property to select frame based outputs.

- The object does not have a property that corresponds to the **Sample time** parameter.

See Also

`comm.PNSequence` | `comm.GoldSequence`

Purpose Create Kasami sequence generator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `KasamiSequence` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.KasamiSequence.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.KasamiSequence.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `KasamiSequence` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.KasamiSequence.reset

Purpose Reset states of Kasami sequence generator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `KasamiSequence` object, `H`.

Purpose

Generate a Kasami sequence

Syntax

```
Y = step(H)
Y = step(H,RESET)
```

Description

`Y = step(H)` outputs a frame of the Kasami sequence in column vector `Y`. Specify the frame length with the `SamplesPerFrame` property. The Kasami sequence has a period of $N = 2^n$-1, where n is the degree of the generator polynomial that you specify in the `Polynomial` property.

`Y = step(H,RESET)` uses `RESET` as the reset signal when you set the `ResetInputPort` property to `true`. The data type of the `RESET` input must be double precision or logical. `RESET` can be a scalar value or a column vector with a length equal to the number of samples per frame that you specify in the `SamplesPerFrame` property. When the `RESET` input is a non-zero scalar, the object resets to the initial conditions that you specify in the `InitialConditions` property. It then generates a new output frame. A column vector `RESET` input allows multiple resets within an output frame. A non-zero value at the i-th element of the vector causes a reset at the i-th output sample time.

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.

---

comm.LDPCDecoder

Purpose

Decode binary low-density parity-check code

Description

The `LDPCDecoder` object decodes a binary low-density parity-check code.

Construction

`h = comm.LDPCDecoder` creates a binary low-density parity-check (LDPC) decoder System object, *h*. This object performs LDPC decoding based on the specified parity-check matrix, where the object does not assume any patterns in the parity-check matrix.

`h = comm.LDPCDecoder('PropertyName','ValueName')` creates an LDPC encoder object, *h*, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`'PropertyName1'`,`'PropertyValue1'`,...,`'PropertyNameN'`,`'PropertyValueN'`).

`h = comm.LDPCDecoder(PARITY)` creates an LDPC decoder object, *h*, with the ParityCheckMatrix property set to PARITY.

Properties

ParityCheckMatrix

Parity-check matrix

Specify the parity-check matrix as a binary valued sparse matrix `P` with dimension (N-by-K) by N, where $N > K > 0$. Alternatively, you can specify a two-column, non-sparse integer index matrix I that defines the row and column indices of the 1s in the parity-check matrix, such that `P = sparse(I(:,1), I(:,2), 1)`.

This property accepts numeric data types. When you set this property to a sparse matrix, it also accepts a logical data type. The upper bound for the value of N is 2^{31}-1.

The default is the sparse parity-check matrix of the half-rate LDPC code from the DVB-S.2 standard, which is the result of `dvbs2ldpc(1/2)`.

To generate code, set this property to a non-sparse index matrix. For instance, you can obtain the index matrix for the DVB-S.2 standard from `dvbs2ldpc(R, 'indices')` with the second input argument explicitly specified to `indices`, where `R` represents the code rate.

OutputValue

Select output value format

Specify the output value format as one of 'Information part' | 'Whole codeword'. The default is 'Information part'. When you set this property to 'Information part', the output contains only the message bits and is a *K* element column vector, assuming an (*N*-by-*K*)x*K* parity check matrix. When you set this property to 'Whole codeword', the output contains the codeword bits and is an *N* element column vector.

DecisionMethod

Decision method

Specify the decision method used for decoding as one of 'Hard decision' | 'Soft decision'. The default is 'Hard decision'. When you set this property to 'Hard decision', the output is decoded bits of double or logical data type. When you set this property to 'Soft decision', the output is log-likelihood ratios of double data type.

IterationTerminationCondition

Condition for iteration termination

Specify the condition to stop the decoding iterations as one of 'Maximum iteration count' | 'Parity check satisfied'. The default is 'Maximum iteration count'. When you set this property to 'Maximum iteration count', the object will iterate for the number of iterations you specify in the `MaximumIterationCount` property. When you set this property to 'Parity check satisfied', the object will determine if the parity checks are satisfied after each iteration and stops if all parity checks are satisfied.

MaximumIterationCount

Maximum number of decoding iterations

Specify the maximum number of iterations the object uses as an integer valued numeric scalar. The default is 50. This applies when you set the `IterationTerminationCondition` property to 'Maximum iteration count'.

NumIterationsOutputPort

Output number of iterations performed

Set this property to true to output the actual number of iterations the object performed. The default is false.

FinalParityChecksOutputPort

Output final parity checks

Set this property to true to output the final parity checks the object calculated. The default is false.

Methods

clone	Create LDPC Decoder object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Encode input using LDPC coding scheme

Examples

Transmit an LDPC-encoded, QPSK-modulated bit stream through an AWGN channel, then demodulate, decode, and count errors.

```
hEnc = comm.LDPCEncoder;
hMod = comm.PSKModulator(4, 'BitInput',true);
hChan = comm.AWGNChannel(...
        'NoiseMethod','Signal to noise ratio (SNR)','SNR',1);
hDemod = comm.PSKDemodulator(4, 'BitOutput',true,...
        'DecisionMethod','Approximate log-likelihood ratio', ...
```

```
        'Variance', 1/10^(hChan.SNR/10));
hDec = comm.LDPCDecoder;
hError = comm.ErrorRate;
for counter = 1:10
  data           = logical(randi([0 1], 32400, 1));
  encodedData    = step(hEnc, data);
  modSignal      = step(hMod, encodedData);
  receivedSignal = step(hChan, modSignal);
  demodSignal    = step(hDemod, receivedSignal);
  receivedBits   = step(hDec, demodSignal);
  errorStats     = step(hError, data, receivedBits);
end
fprintf('Error rate       = %1.2f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the LDPC Decoder block reference page. The object properties correspond to the block parameters.

See Also

comm.LDPCEncoder | comm.BCHDecoder | comm.gpu.LDPCDecoder

comm.LDPCDecoder.clone

Purpose Create LDPC Decoder object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates an `LDPC Decoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `LDPCEncode` 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.

comm.LDPCDecoder.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 Encode input using LDPC coding scheme

Syntax

```
Y = step(H,X)
[Y,NUMITER] = step(H,X)
[Y,PARITY] = step(H,X)
```

Description

`Y = step(H,X)` decodes input codeword, `X`, using an LDPC code that is based on an (*N*-*K*) x *N* parity-check matrix. You specify the parity-check matrix in the `ParityCheckMatrix` property. Input `X` must be a double column vector with length equal *N*. Each element is the log-likelihood ratio for a received bit (more likely to be 0 if the log-likelihood ratio is positive). The first *K* elements correspond to the information part of a codeword. The decoded data output vector, `Y`, contains either only the message bits or the whole code word, based on the value of the `OutputValue` property.

`[Y,NUMITER] = step(H,X)` returns the actual number of iterations the object performed when you set the `NumIterationsOutputPort` property to true. The step method outputs `NUMITER` as a double scalar.

`[Y,PARITY] = step(H,X)` returns final parity checks the object calculated when you set the `FinalParityChecksOutputPort` property to true. The step method outputs `PARITY` as a double vector of length (*N*-*K*). You can combine optional output arguments when you set their enabling properties. Optional outputs must be listed in the same order as the order of the enabling properties. For example, `[Y,NUMITER,PARITY] = step(H,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

Encode binary low-density parity-check code

Description

The `LDPCEncoder` object encodes a binary low-density parity-check code.

Construction

`h = comm.LDPCEncoder` creates a binary low-density parity-check (LDPC) encoder System object, *h*. This object performs LDPC encoding based on the specified parity-check matrix.

`h = comm.LDPCEncoder('PropertyName','ValueName')` creates an LDPC encoder object, *h*, with each specified property set to the specified value.

`h = comm.LDPCEncoder(PARITY)` creates an LDPC encoder object, *h*, with the ParityCheckMatrix property set to PARITY.

`output_args = function(input_args,Name,Value)` with additional options specified by one or more `Name,Value` pair arguments. `Name` can also be a property name and `Value` is the corresponding value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

Properties

ParityCheckMatrix

Parity-check matrix

Specify the parity-check matrix as a binary valued sparse matrix `P` with dimension (N-by-K) by N, where $N > K > 0$. Alternatively, you can specify a two-column, non-sparse integer index matrix I that defines the row and column indices of the 1s in the parity-check matrix, such that `P = sparse(I(:,1), I(:,2), 1)`.

This property accepts numeric data types. When you set this property to a sparse matrix, it also accepts a logical data type. The upper bound for the value of N is 2^{31}-1.

The default is the sparse parity-check matrix of the half-rate LDPC code from the DVB-S.2 standard, which is the result of `dvbs2ldpc(1/2)`.

To generate code, set this property to a non-sparse index matrix. For instance, you can obtain the index matrix for the DVB-S.2 standard from `dvbs2ldpc(R, 'indices')` with the second input argument explicitly specified to `indices`, where `R` represents the code rate.

Methods

clone	Create LDPC Encoder object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Encode input using LDPC coding scheme

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects in the MATLAB documentation.

Examples

Transmit an LDPC-encoded, QPSK-modulated bit stream through an AWGN channel, then demodulate, decode, and count errors.

```
hEnc = comm.LDPCEncoder;
hMod = comm.PSKModulator(4, 'BitInput',true);
hChan = comm.AWGNChannel(...
        'NoiseMethod','Signal to noise ratio (SNR)','SNR',1);
hDemod = comm.PSKDemodulator(4, 'BitOutput',true,...
        'DecisionMethod','Approximate log-likelihood ratio', ...
        'Variance', 1/10^(hChan.SNR/10));
hDec = comm.LDPCDecoder;
hError = comm.ErrorRate;
for counter = 1:10
  data          = logical(randi([0 1], 32400, 1));
  encodedData    = step(hEnc, data);
  modSignal      = step(hMod, encodedData);
```

```
    receivedSignal = step(hChan, modSignal);
    demodSignal    = step(hDemod, receivedSignal);
    receivedBits   = step(hDec, demodSignal);
    errorStats     = step(hError, data, receivedBits);
  end
  fprintf('Error rate       = %1.2f\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the LDPC Encoder block reference page. The object properties correspond to the block parameters.

See Also

comm.LDPCDecoder | comm.BCHEncoder

Purpose Create LDPC Encoder object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates an `LDPC Encoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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.

comm.LDPCEncoder.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `LDPCEncode` 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)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.LDPCEncoder.step

Purpose

Encode input using LDPC coding scheme

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` encodes input binary message, `X`, using an LDPC code that is based on an (N-K) x N parity-check matrix. You specify the parity-check matrix in the `ParityCheckMatrix` property. Input `X` must be a numeric or logical column vector with length equal K. The length of the encoded data output vector, `Y`, is N. It is a solution to the parity-check equation, with the first K bits equal to 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

Filter input signal through LTE MIMO multipath fading channel

Description

The `comm.LTEMIMOChannel` System object filters an input signal through an LTE multiple-input multiple-output (MIMO) multipath fading channel.

A specialization of the `comm.MIMOChannel` System object, the `comm.LTEMIMOChannel` System objects offers pre-set configurations for use with LTE link level simulations. In addition to the `comm.MIMOChannel` System object, the `comm.LTEMIMOChannel` System object also corrects the correlation matrix to be positive semi-definite, after rounding to 4-digit precision. This System object models Rayleigh fading for each of its links.

Construction

`H = comm.LTEMIMOChannel` creates a 3GPP Long Term Evolution (LTE) Release 10 specified multiple-input multiple-output (MIMO) multipath fading channel System object, `H`. This object filters a real or complex input signal through the multipath LTE MIMO channel to obtain the channel impaired signal.

`H = comm.LTEMIMOChannel(Name,Value)` creates an LTE MIMO multipath fading channel 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

Input signal sample rate (Hertz)

Specify the sample rate of the input signal in hertz as a double-precision, real, positive scalar. The default value of this property is `30.72` MHz, as defined in the LTE specification.

Profile

Channel propagation profile

Specify the propagation conditions of the LTE multipath fading channel as one of `EPA 5 Hz | EVA 5 Hz | EVA 70 Hz | ETU 70`

`Hz | ETU 300 Hz`, which are supported in the LTE specification Release 10. The default value of this property is `EPA 5 Hz`.

This property defines the delay profile of the channel to be one of EPA, EVA, and ETU. This property also defines the maximum Doppler shift of the channel to be 5 Hz, 70 Hz, or 300 Hz. The Doppler spectrum always has a Jakes shape in the LTE specification. The EPA profile has seven paths. The EVA and ETU profiles have nine paths.

The following tables list the delay and relative power per path associated with each profile.

Extended Pedestrian A Model (EPA)

Excess tap delay [ns]	Relative power [db]
0	0.0
30	-1.0
70	-2.0
90	-3.0
110	-8.0
190	-17.2
410	-20.8

Extended Vehicular A Model (EVA)

Excess tap delay [ns]	Relative power [db]
0	0.0
30	-1.5
150	-1.4
310	-3.6
370	-0.6

Excess tap delay [ns]	Relative power [db]
710	-9.1
1090	-7.0
1730	-12.0
2510	-16.9

Extended Typical Urban Model (ETU)

Excess tap delay [ns]	Relative power [db]
0	-1.0
50	-1.0
120	-1.0
200	0.0
230	0.0
500	0.0
1600	-3.0
2300	-5.0
5000	-7.0

AntennaConfiguration

Antenna configuration

Specify the antenna configuration of the LTE MIMO channel as one of `1x2` | `2x2` | `4x2` | `4x4`. These configurations are supported in the LTE specification Release 10. The default value of this property is `2x2`.

The property value is in the format of N_t-by-N_r. N_t represents the number of transmit antennas and N_r represents the number of receive antennas.

CorrelationLevel

Spatial correlation strength

Specify the spatial correlation strength of the LTE MIMO channel as one of `Low | Medium | High`. The default value of this property is `Low`. When you set this property to `Low`, the MIMO channel is spatially uncorrelated.

The transmit and receive spatial correlation matrices are defined from this property according to the LTE specification Release 10. See the Algorithms section for more information.

AntennaSelection

Antenna selection

Specify the antenna selection scheme as one of `Off | Tx | Rx | Tx and Rx`, where `Tx` represents transmit antennas and `Rx` represents receive antennas. When you select `Tx` and/or `Rx`, additional input(s) are required to specify which antennas are selected for signal transmission. The default value of this property is `Off`.

RandomStream

Source of random number stream

Specify the source of random number stream as one of `Global stream | mt19937ar with seed`. The default value of this property is `Global stream`. When you set this property to `Global stream`, the current global random number stream is used for normally distributed random number generation. In this case, the `reset` method only resets the filters. If you set `RandomStream` to `mt19937ar with seed`, the object uses the mt19937ar algorithm for normally distributed random number generation. In this case, the `reset` method resets the filters and reinitializes the random number stream to the value of the `Seed` property.

Seed

Initial seed of mt19937ar random number stream

Specify the initial seed of an mt19937ar random number generator algorithm as a double-precision, real, nonnegative integer scalar.

The default value of this property is `73`. This property applies when you set the `RandomStream` property to `mt19937ar with seed`. The `Seed` reinitializes the mt19937ar random number stream in the `reset` method.

NormalizePathGains

Normalize path gains (logical)

Set this property to `true` to normalize the fading processes so that the total power of the path gains, averaged over time, is `0` dB. The default value of this property is `true`. When you set this property to `false`, there is no normalization for path gains.

NormalizeChannelOutputs

Normalize channel outputs (logical)

Set this property to `true` to normalize the channel outputs by the number of receive antennas. The default value of this property is `true`. When you set this property to `false`, there is no normalization for channel outputs.

PathGainsOutputPort

Enable path gain output (logical)

Set this property to `true` to output the channel path gains of the underlying fading process. The default value of this property is `false`.

Methods

clone	Create object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the object
step	Filter input signal through LTE MIMO multipath fading channel

Examples

Configure Equivalent MIMO Channel System Object Using an LTE MIMO Channel System Object

Configure an equivalent `MIMOChannel` System Object using the `LTEMIMOChannel` System Object. Then, verify that the channel output and the path gain output from the two objects are the same.

Create a PSK Modulator System object to modulate randomly generated data.

```
hMod = comm.PSKModulator;
modData = step(hMod, randi([0 hMod.ModulationOrder-1],2e3,1));
```

Split modulated data into two spatial streams.

```
channelInput = reshape(modData, [2, 1e3]).';
```

Create an `LTEMIMOChannel` System object with a 2-by-2 antenna configuration and a medium correlation level.

```
hLTEChan = comm.LTEMIMOChannel(...
  'Profile',              'EVA 5Hz',...
  'AntennaConfiguration', '2x2',...
  'CorrelationLevel',     'Medium',...
  'RandomStream',         'mt19937ar with seed',...
  'Seed',                 99,...
  'PathGainsOutputPort',  true);
```

Filter the modulated data using the `LTEMIMOChannel` System object, `hLTEChan`.

```
[LTEChanOut, LTEPathGains] = step(hLTEChan,  channelInput);
```

Create an equivalent `MIMOChannel` System object, `hMIMOChan`, using the properties of the `LTEMIMOChannel` System object, `hLTEChan`.

Note The `KFactor`, `DirectPathDopplerShift` and `DirectPathInitialPhase` properties only exist for the `MIMOChannel` System object. All other `MIMOChannel` System object properties also exist for the `LTEMIMOChannel` System object; however, some properties are hidden and read-only.

```
hMIMOChan = comm.MIMOChannel(...
  'SampleRate',                hLTEChan.SampleRate,...
  'PathDelays',                hLTEChan.PathDelays,...
  'AveragePathGains',          hLTEChan.AveragePathGains,...
  'MaximumDopplerShift',       hLTEChan.MaximumDopplerShift,...
  'DopplerSpectrum',           hLTEChan.DopplerSpectrum,...
  'NumTransmitAntennas',       hLTEChan.NumTransmitAntennas,...
  'NumReceiveAntennas',        hLTEChan.NumReceiveAntennas,...
  'TransmitCorrelationMatrix', hLTEChan.TransmitCorrelationMatrix,...
  'ReceiveCorrelationMatrix',  hLTEChan.ReceiveCorrelationMatrix,...
  'FadingDistribution',        hLTEChan.FadingDistribution,...
  'RandomStream',              hLTEChan.RandomStream,...
  'Seed',                      hLTEChan.Seed,...
  'NormalizePathGains',        hLTEChan.NormalizePathGains,...
  'NormalizeChannelOutputs',   hLTEChan.NormalizeChannelOutputs,...
  'PathGainsOutputPort',       hLTEChan.PathGainsOutputPort);
```

Filter the modulated data using the equivalent `hMIMOChan` and use the `step` method to process data.

```
[MIMOChanOut, MIMOPathGains] = step(hMIMOChan, channelInput);
```

Verify that the channel output and the path gain output from the two objects are the same.

```
display(isequal(LTEChanOut, MIMOChanOut));
display(isequal(LTEPathGains, MIMOPathGains));
```

Note You can repeat the preceding process with `AntennaConfiguration` set to `4x2` or `4x4` and `CorrelationLevel` set to `Medium` or `High` for `hLTEChan`. If you do so, the resulting channel output and path gain output from the two objects are slightly different. This difference occurs because an LTE channel with such configurations has its spatial correlation matrix rounded to 4-digit precision. See the LTE specification Release 10 for more details.

Algorithms

This System object is a specialized implementation of the `comm.MIMOChannel` System object. For additional algorithm information, see the `comm.MIMOChannel` System object help page.

Spatial Correlation Matrices

The following table defines the transmitter eNodeB correlation matrix.

	One Antenna	Two Antennas	Four Antennas
eNodeB Correlation	$R_{\mathrm{eNB}} = 1$	$R_{eNB} = \begin{pmatrix} 1 & \alpha \\ \alpha^* & 1 \end{pmatrix}$	$R_{eNB} = \begin{pmatrix} 1 & \alpha^{1/9} & \alpha^{4/9} \\ \alpha^{1/9 *} & 1 & \alpha^{1/9} \\ \alpha^{4/9 *} & \alpha^{1/9 *} & 1 \\ \alpha^{*} & \alpha^{4/9 *} & \alpha^{1/9 *} \end{pmatrix}$

The following table defines the receiver UE correlation matrix.

	One Antenna	Two Antennas	Four Antennas
UE Correlation	$R_{\mathrm{UE}} = 1$	$R_{UE} = \begin{pmatrix} 1 & \beta \\ \beta^* & 1 \end{pmatrix}$	$R_{UE} = \begin{pmatrix} 1 & \beta^{1/9} & \beta^{4/9} \\ \beta^{1/9^*} & 1 & \beta^{1/9} \\ \beta^{4/9^*} & \beta^{1/9^*} & 1 \\ \beta^* & \beta^{4/9^*} & \beta^{1/9} \end{pmatrix}$

The following table describes the R_{spat} channel spatial correlation matrix between the transmitter and receiver antennas.

Tx-by-Rx Configuration	Correlation Matrix
1-by-2	$R_{spat} = R_{UE} = \begin{bmatrix} 1 & \beta \\ \beta^* & 1 \end{bmatrix}$
2-by-2	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha \\ \alpha^* & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & \beta \\ \beta^* & 1 \end{bmatrix} =$

Tx-by-Rx Configuration	Correlation Matrix
4-by-2	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha^{1/9} & \alpha^{4/9} & \alpha \\ \alpha^{1/9^{*}} & 1 & \alpha^{1/9} & \alpha^{4/9} \\ \alpha^{4/9^{*}} & \alpha^{1/9^{*}} & 1 & \alpha^{1/9} \\ \alpha^{*} & \alpha^{4/9^{*}} & \alpha^{1/9^{*}} & 1 \end{bmatrix}$
4-by-4	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha^{1/9} & \alpha^{4/9} & \alpha \\ \alpha^{1/9^{*}} & 1 & \alpha^{1/9} & \alpha^{4/9} \\ \alpha^{4/9^{*}} & \alpha^{1/9^{*}} & 1 & \alpha^{1/9} \\ \alpha^{*} & \alpha^{4/9^{*}} & \alpha^{1/9^{*}} & 1 \end{bmatrix}$

Spatial Correlation Correction

Low Correlation		Medium Correlation		High Correlation	
α	ß	α	ß	α	ß
0	0	0.3	0.9	0.9	0.9

To insure the correlation matrix is positive semi-definite after round-off to 4 digit precision, this System object uses the following equation:

$$R_{high} = \left[R_{spatial} + aI_n\right] / (1 + a)$$

Where

α represents the scaling factor such that the smallest value is used to obtain a positive semi-definite result.

For the 4-by-2 high correlation case, α=0.00010.

For the 4-by-4 high correlation case, α=0.00012.

The object uses the same method to adjust the 4-by-4 medium correlation matrix to insure the correlation matrix is positive semi-definite after rounding to 4 digit precision with α = 0.00012.

Selected Bibliography

[1] 3rd Generation Partnership Project, Technical Specification Group Radio Access Network, Evolved Universal Terrestrial Radio Access (E-UTRA), *Base Station (BS) radio transmission and reception*, Release 10, 2009–2010, 3GPP TS 36.104, Vol. 10.0.0.

[2] 3rd Generation Partnership Project, Technical Specification Group Radio Access Network, Evolved Universal Terrestrial Radio Access (E-UTRA), *User Equipment (UE) radio transmission and reception*, Release 10, 2010, 3GPP TS 36.101, Vol. 10.0.0.

[3] Oestges, C., and B. Clerckx. *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*, Academic Press, 2007.

[4] Correira, L. M. *Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G*, Academic Press, 2006.

[5] Jeruchim, M., P. Balaban, and K. S. Shanmugan. *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

See Also

`comm.MIMOChannel`

comm.LTEMIMOChannel.clone

Purpose Create LTEMIMOChannel object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates an `LTEMIMOChannel` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`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`*`)`.

comm.LTEMIMOChannel.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

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

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `LTEMIMOChannel` 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 or hardware connections) and allows all properties and input characteristics to be changed.

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 the `LTEMIMOChannel` object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `LTEMIMOChannel` object, `H`.

If you set the `RandomStream` property of `H` to `Global stream`, the `reset` method only resets the filters. If you set `RandomStream` to `mt19937ar with seed`, the `reset` method not only resets the filters but also reinitializes the random number stream to the value of the `Seed` property.

comm.LTEMIMOChannel.step

Purpose

Filter input signal through LTE MIMO multipath fading channel

Syntax

```
Y = step(H,X)
[Y,PATHGAINS] = step(H,X)
```

Description

`Y = step(H,X)` filters input signal `X` through an LTE MIMO multipath fading channel and returns the result in `Y`. The input `X` can be a double-precision data type scalar, vector, or 2-D matrix with real or complex values. `X` is of size N_s-by-N_t. N_s represents the number of samples and N_t represents the number of transmit antennas that must match the `AntennaConfiguration` property setting of `H`. `Y` is the output signal of size N_s-by-N_r. N_r represents the number of receive antennas that is specified by the `AntennaConfiguration` property of `H`. `Y` is of double-precision data type with complex values.

`[Y,PATHGAINS] = step(H,X)` returns the LTE MIMO channel path gains of the underlying fading process in PATHGAINS. This applies when you set the `PathGainsOutputPort` property to `true`. PATHGAINS is of size N_s-by-N_p-by-N_t-by-N_r. N_p represents the number of discrete paths of the channel implicitly defined by the `Profile` property of `H`. PATHGAINS is of double-precision data type with complex values.

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

Deinterleave input symbols using permutation matrix

Description

The `MatrixDeinterleaver` object performs block deinterleaving by filling a matrix with the input symbols column by column and then sending the matrix contents to the output port row by row. The number of rows and number of columns properties set the dimensions of the matrix that the object uses internally for computations.

Construction

`H = comm.MatrixDeinterleaver` creates a matrix deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using the matrix interleaver object.

`H = comm.MatrixDeinterleaver(Name,Value)` creates a matrix deinterleaver 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`).

`H = comm.MatrixDeinterleaver(N,M)` creates a matrix deinterleaver object, `H`. This object has the `NumRows` property set to `N`, the `NumColumns` property set to `M`.

Properties

NumRows

Number of rows of permutation matrix

Specify the number of permutation matrix rows as a scalar, positive integer. The default is `3`.

NumColumns

Number of columns of permutation matrix

Specify the number of permutation matrix columns as a scalar, positive integer. The default is `4`.

Methods

clone	Create matrix deinterleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Deinterleave input symbols using permutation matrix

Examples

Interleave and deinterleave data.

```
hInt = comm.MatrixInterleaver('NumRows', 2, ...
                'NumColumns', 5);
hDeInt = comm.MatrixDeinterleaver('NumRows', 2, ...
                'NumColumns', 5);
data = randi(7, 10, 1);
intData = step(hInt, data);
deIntData = step(hDeInt, intData);
[data, intData, deIntData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Matrix Deinterleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.MatrixInterleaver` | `comm.BlockDeinterleaver`

Purpose Create matrix deinterleaver object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `MatrixDeinterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.MatrixDeinterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.MatrixDeinterleaver.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `MatrixDeinterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.MatrixDeinterleaver.step

Purpose

Deinterleave input symbols using permutation matrix

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a block interleaver. The object fills a permutation matrix with the input symbols column by column and outputs the matrix contents row by row in the output, `Y`. The input `X` must be a column vector of length equal to `NumRows`×`NumColumns`. The data type for `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `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

Permute input symbols using permutation matrix

Description

The `MatrixInterleaver` object performs block interleaving by filling a matrix with the input symbols row by row and then outputs the matrix contents column-by-column.

Construction

`H = comm.MatrixInterleaver` creates a matrix interleaver System object, `H`. This object permutes the input by filling a permutation matrix with the input symbols row by row. The object then outputs the matrix contents column by column.

`H = comm.MatrixInterleaver(Name,Value)` creates a matrix interleaver 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`).

`H = comm.MatrixInterleaver(N,M)` creates a matrix interleaver object, `H`. This object has the `NumRows` property set to `N`, the `NumColumns` property set to `M`.

Properties

NumRows

Number of rows of permutation matrix

Specify the number of permutation matrix rows as a scalar, positive integer. The default is `3`.

NumColumns

Number of columns of permutation matrix

Specify the number of permutation matrix columns as a scalar, positive integer. The default is `4`.

Methods

clone	Create matrix interleaver object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Permute input symbols using permutation matrix

Examples

Interleave and deinterleave data

```
hInt = comm.MatrixInterleaver('NumRows', 2, ...
                'NumColumns', 5);
hDeInt = comm.MatrixDeinterleaver('NumRows', 2, ...
                'NumColumns', 5);
data = randi(7, 10, 1);
intData = step(hInt, data);
deIntData = step(hDeInt, intData);

% compare the original sequence, interleaved sequence, and restored s
[data, intData, deIntData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Matrix Deinterleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.MatrixDeinterleaver` | `comm.BlockInterleaver`

Purpose

Create matrix interleaver object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `MatrixInterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.MatrixInterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.MatrixInterleaver.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `MatrixInterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.MatrixInterleaver.step

Purpose

Permute input symbols using permutation matrix

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The object fills a permutation matrix with the input symbols row by row and outputs the matrix contents column by column. The input `X` must be a column vector of length `NumRows`×`NumColumns` and the data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `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

Deinterleave input symbols by filling a matrix along diagonals

Description

The `MatrixHelicalScanDeinterleaver` object performs block deinterleaving by filling a matrix with the input symbols helically and then outputs the matrix contents row by row. The number of rows and number of columns properties represent the dimensions of the matrix that the object uses internally for computations.

Construction

`H = comm.MatrixHelicalScanDeinterleaver` creates a matrix helical scan deinterleaver object, `H`. This object restores the original ordering of a sequence that was interleaved using the matrix helical scan interleaver System object.

`H = comm.MatrixHelicalScanDeinterleaver(Name,Value)` creates a matrix helical scan deinterleaver 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

NumRows

Number of rows of permutation matrix

Specify the number of rows in the permutation matrix as a scalar, positive integer. The default is `64`.

NumColumns

Number of columns of permutation matrix

Specify the number of columns in the permutation matrix as a scalar, positive integer. The default is `64`.

StepSize

Slope of diagonals

Specify slope as a scalar integer between `0` and the value you specify in the `NumRows` property. The default is `1`. The slope value indicates the amount by which the row index increases as the column index increases by 1. When you set the value of this property to 0, the object does not interleave and the output matches the input.

Methods

clone	Create matrix helical scan deinterleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Deinterleave input symbols by filling a matrix along diagonals

Examples

Interleave and deinterleave random data.

```
hInt = comm.MatrixHelicalScanInterleaver('NumRows', 4, ...
                'NumColumns', 4);
hDeInt = comm.MatrixHelicalScanDeinterleaver('NumRows', 4, ...
                'NumColumns', 4);
data = randi(7, 16, 1);
intData = step(hInt, data);
deIntData = step(hDeInt, intData);

% compare the original sequence, interleaved sequence and restored se
[data, intData, deIntData];
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Matrix Helical Scan Deinterleaver block reference page. The object properties correspond to the block parameters.

See Also

comm.MatrixHelicalScanInterleaver | comm.BlockDeinterleaver

comm.MatrixHelicalScanDeinterleaver.clone

Purpose

Create matrix helical scan deinterleaver object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `MatrixHelicalScanDeinterleaver` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.MatrixHelicalScanDeinterleaver.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `MatrixHelicalScanDeinterleaver` 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.

comm.MatrixHelicalScanDeinterleaver.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Deinterleave input symbols by filling a matrix along diagonals

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` restores the original ordering of the sequence, `X`. The object fills a permutation matrix with the input symbols in a helical fashion and output the contents row by row, and returns `Y`. The input `X` must be a `NumRows`×`NumColumns` long column vector and the data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `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.

---

comm.MatrixHelicalScanInterleaver

Purpose

Permute input symbols by selecting matrix elements along diagonals

Description

The `MatrixHelicalScanInterleaver` object performs block interleaving by filling a matrix with the input symbols row by row and then outputs the matrix contents in a helical helically. The number of rows and number of columns properties are the dimensions of the matrix that the object uses internally for computations.

Construction

`H = comm.MatrixHelicalScanInterleaver` creates a matrix helical scan interleaver object, `H`. This object permutes the input by filling a permutation matrix with the input symbols row by row and then outputs the matrix contents helically.

`H = comm.MatrixHelicalScanInterleaver(Name,Value)` creates a matrix helical scan interleaver 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

NumRows

Number of rows of permutation matrix

Specify the number of rows in the permutation matrix as a scalar, positive integer. The default is `64`.

NumColumns

Number of columns of permutation matrix

Specify the number of columns in the permutation matrix as a scalar, positive integer. The default is `64`.

StepSize

Slope of diagonals

Specify slope as a scalar integer between 0 and the value you specify in the `NumRows` property. The slope value represents the amount by which the row index increases as the column index increases by 1. When you set the value of this property to `0`, the object does not interleave and the output matches the input. The default is 1.

Methods

clone	Create matrix helical scan interleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Permute input symbols by selecting matrix elements along diagonals

Examples

Interleave and deinterleave random data.

```
hInt = comm.MatrixHelicalScanInterleaver('NumRows', 4, ...
                'NumColumns', 4);
hDeInt = comm.MatrixHelicalScanDeinterleaver('NumRows', 4, ...
                'NumColumns', 4);
data = randi(7, 16, 1);
intData = step(hInt, data);
deIntData = step(hDeInt, intData);

% compare the original sequence, interleaved sequence and restored
[data, intData, deIntData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Matrix Helical Scan Deinterleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.MatrixHelicalScanDeinterleaver` | `comm.BlockInterleaver`

Purpose

Create matrix helical scan interleaver object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `MatrixHelicalScanInterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.MatrixHelicalScanInterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.MatrixHelicalScanInterleaver.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `MatrixHelicalScanInterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.MatrixHelicalScanInterleaver.step

Purpose Permute input symbols by selecting matrix elements along diagonals

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The input `X` must be a `NumRows`×`NumColumns` long column vector and the data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `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

Apply memoryless nonlinearity to input signal

Description

The `MemorylessNonlinearity` object applies a memoryless nonlinearity to a complex, baseband signal. You can use the object to model radio frequency (RF) impairments to a signal at the receiver.

Construction

`H = comm.MemorylessNonlinearity` creates a memoryless nonlinearity System object, `H`. This object models receiver radio frequency (RF) impairments.

`H = comm.MemorylessNonlinearity(Name,Value)` creates a memoryless nonlinearity 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

Method

Method used to model nonlinearity

Specify the nonlinearity method as one of `Cubic polynomial | Hyperbolic tangent | Saleh model | Ghorbani model | Rapp model`. The default is `Cubic polynomial`. This property is non-tunable.

InputScaling

Scale factor applied to input signal

Specify the scale factor in decibels. The object applies this factor to the input signal as a real scalar value of double- or single-precision data type. The default is `0`. This property applies when you set the `Method` property to `Saleh model` or `Ghorbani model`. This property is tunable.

LinearGain

Linear gain applied to output signal

Specify the linear gain (in decibels) that the object applies to the output signal as a real scalar value of double- or single-precision data type. The default is `0`. This property applies when you set the Method property to Cubic polynomial, Hyperbolic tangent, or Rapp model. This property is tunable.

IIP3

Third-order input intercept point

Specify the third-order input intercept point (in decibels relative to a milliwatt) as a real scalar value of double- or single-precision data type. The default is 30. This property applies when you set the Method property to Cubic polynomial or Hyperbolic tangent. This property is tunable.

AMPMConversion

AM/PM conversion factor

Specify the AM/PM conversion factor (in degrees per decibel) as a real scalar value of double- or single-precision data type. The default is 10. This property applies when you set the Method property to Cubic polynomial or Hyperbolic tangent. This property is tunable.

AMAMParameters

AM/AM conversion parameters

Specify the AM/AM conversion parameters that the object uses to compute the amplitude gain for an input signal as a real vector of double- or single-precision data type. The default is [2.1587 1.1517] for the Saleh model and [8.1081 1.5413 6.5202 -0.0718] for the Ghorbani model.

This property applies when you set the Method property to Saleh model or Ghorbani model.

When you set the Method property to Saleh model, this property is a two-element vector that specifies alpha and beta values. Otherwise, this property is a four-element vector that specifies x1, x2, x3, and x4 values. This property is tunable.

AMPMParameters

AM/PM conversion parameters

Specify the AM/PM conversion parameters used to compute the phase change for an input signal as a real vector of double- or single-precision data type. The default is `[4.0033 9.1040]` for the Saleh model and `[4.6645 2.0965 10.88 -0.003]` for the Ghorbani model.

This property applies when you set the `Method` property to `Saleh model` or `Ghorbani model`.

When you set the `Method` property to `Saleh model`, this property is a two-element vector that specifies alpha and beta values. Otherwise, this property is a four-element vector that specifies y1, y2, y3, and y4 values. This property is tunable.

PowerLowerLimit

Lower input power limit

Specify the minimum input power (in decibels relative to a milliwatt) for which AM/PM conversion scales linearly with input power value. The default is `10`. Below this value, the phase shift resulting from AM/PM conversion is zero. You must set this property to a real scalar value of double- or single-precision data type. This property applies when you set the `Method` property to `Cubic polynomial` or `Hyperbolic tangent`. This property is tunable.

PowerUpperLimit

Upper input power limit

Specify the maximum input power (in decibels relative to a milliwatt) for which AM/PM conversion scales linearly with input power value. The default is `inf`. Above this value, the phase shift resulting from AM/PM conversion is constant. You must set the `PowerUpperLimit` property to a real scalar value, which is greater than the `PowerLowerLimit` property and of double- or single-precision data type. This property applies when you set the `Method` property to `Cubic polynomial` or `Hyperbolic tangent`.This property is tunable.

OutputScaling

Scale factor applied to output signal

Specify the scale factor (in decibels) that the object applies to the output signal as a real scalar value of double- or single-precision data type. The default is `0`. This property applies when you set the `Method` property to `Saleh model` or `Ghorbani model`. This property is tunable.

Smoothness

Smoothness factor

Specify the smoothness factor as a real scalar value of double- or single-precision data type. The default is `0.5`. This property applies when you set the `Method` property to `Rapp model`. This property is tunable.

OutputSaturationLevel

Output saturation level

Specify the output saturation level as a real scalar value of double- or single-precision data type. This property applies when you set the `Method` property to `Rapp model`. The default is `1`. This property is tunable.

Methods

clone	Create memoryless nonlinearity object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Apply memoryless nonlinearity to input signal

Examples

Apply "Saleh model" nonlinearity to a 16-QAM signal.

```
% Create 16-QAM modulator
hMod = comm.RectangularQAMModulator('ModulationOrder',16,...
            'NormalizationMethod','Average power', 'AveragePower',
% Create Memoryless Nonlinearity System object
hNonlin = comm.MemorylessNonlinearity('Method', 'Saleh model');
% Generate modulated symbols
modData = step(hMod, randi([0 15], 1000, 1));
% Apply Nonlinearity model and plot the result
y = step(hNonlin, modData);
scatterplot(y)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Memoryless Nonlinearity block reference page. The object properties correspond to the block parameters.

See Also

`comm.PhaseNoise`

comm.MemorylessNonlinearity.clone

Purpose Create memoryless nonlinearity object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `MemorylessNonlinearity` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.MemorylessNonlinearity.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `MemorylessNonlinearity` 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.

comm.MemorylessNonlinearity.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Apply memoryless nonlinearity to input signal

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` applies memoryless nonlinearity to the input, `X`, using the nonlinearity method you specify in the `Method` property, and returns the result in `Y`. The input `X` must be a complex scalar or column vector of data type double or single precision. The output, `Y`, is of the same data type as 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

Measure modulation error ratio

Description

The Modulation Error Ratio (MER) `MER` object measures the signal-to-noise ratio (SNR) in digital modulation applications. You can use these types of measurements to determine system performance in communications applications. For example, determining if an EDGE system conforms to 3GPP radio transmission standards requires accurate MER, Minimum MER, and 95th percentile for the MER measurements. The block measures all outputs in decibels.

Construction

`H = comm.MER` creates a modulation error ratio (MER) System object, `H`. This object measures the signal-to-noise ratio (SNR) in digital modulation applications.

`H = comm.MER(Name,Value)` creates an MER 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

MinimumMEROutputPort

Enable minimum MER measurement output

When you set this property to `true`, the `step` method outputs minimum MER measurements. The default is `false`. The `step` method outputs the minimum MER output as the minimum MER value measured in the current input frame.

XPercentileMEROutputPort

Enable X-percentile MER measurement output

When you set this property to `true`, the `step` method outputs X-percentile MER measurements. The default is `false`. The X-percentile MER measurements persist. These measurements are based on all the input frames since the last reset.

XPercentileValue

X-percentile value

Specify the X-percentile value (as a percentage) that the object uses to calculate the x^{th} percentile of the MER measurements. The default is 95. Set this property to a real scalar value between 0 and 100, inclusive. This property can have a data type of double, single, or integer. This property applies when you set the XPercentileMEROutputPort property to true. The x-th percentile is the MER value above which x% of all the computed MER values lie.

SymbolCountOutputPort

Enable symbol count output

When you set this property to true, the step method outputs the number of accumulated symbols that have been used to calculate the x-Percentile MER measurements since the last reset. The default is false. This property applies when you set the XPercentileMEROutputPort property to true.

Methods

clone	Create MER measurement object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of MER measurement object
step	Measure modulation error ratio

Examples

Measure MER of a noisy 16-QAM modulated signal.

```
   hMod = comm.RectangularQAMModulator(16);
   hAWGN = comm.AWGNChannel('NoiseMethod',...
                'Signal to noise ratio (SNR)',...
                'SNR', 20, 'SignalPower', 10);
% Create an MER object, output minimum and 90-percentile MER, and symbol
% count
   hMER = comm.MER('MinimumMEROutputPort', true, ...
                'XPercentileMEROutputPort', true,'XPercentileValue', 90,
                'SymbolCountOutputPort',true);
% Generate modulated symbols and add noise
   refsym = step(hMod, randi([0 15], 1000, 1));
   rxsym = step(hAWGN, refsym);
% Calculate measurements
   [MERdB,MinMER,PercentileMER,NumSym] = step(hMER,refsym,rxsym)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the MER Measurement block reference page. The object properties correspond to the block parameters.

See Also

comm.EVM | comm.ACPR | comm.CCDF

Purpose Create MER measurement object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `MER` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.MER.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.MER.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `MER` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.MER.reset

Purpose Reset states of MER measurement object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `MER` object, `H`.

Purpose

Measure modulation error ratio

Syntax

```
MERDB = step(H,REFSYM,RXSYM)
[MERDB,MINMER] = step(H,REFSYM,RXSYM)
[MERDB,PMER] = step(H,REFSYM,RXSYM)
[MERDB,NUMSYM] = step(H,REFSYM,RXSYM)
```

Description

`MERDB = step(H,REFSYM,RXSYM)` outputs MER (in dB), `MERDB`, measured in the received signal, `RXSYM`, based on the reference signal, `REFSYM`. `REFSYM`, and `RXSYM` inputs are complex column vectors of equal dimensions and data type. The data type can be double, single, signed integer, or signed fixed point with power-of-two slope and zero bias. The step method outputs the `MERDB` measurement based solely on the current input frame. All outputs of this object are of data type double.

`[MERDB,MINMER] = step(H,REFSYM,RXSYM)` outputs the minimum MER (in dB), `MINMER`, measured in the received signal, `RXSYM`, when you set the `MinimumMEROutputPort` property to true. The step method outputs the `MINMER` measurement based on the reference signal, `REFSYM`. `MINMER` is the minimum MER value measured in the current input frame.

`[MERDB,PMER] = step(H,REFSYM,RXSYM)` outputs the percentile MER (in dB), `PMER`, measured in the received signal, `RXSYM`, when you set the `XPercentileMEROutputPort` property to true. The step method outputs the `PMER` measurement based on the reference signal, `REFSYM`. The object sets `PMER` equal to a value just smaller than the XPercentileValue percent of all the MER values. For example, if you set the `XPercentileValue` property to 95, then 95% of all MER measurements are above the `PMER` value. The object calculates the persistent measurement `PMER`, using all the input frames since the last reset.

`[MERDB,NUMSYM] = step(H,REFSYM,RXSYM)` outputs the number of symbols, `NUMSYM`, used to calculate the X-Percentile MER measurements when you set the `SymbolCountOutputPort` property to true. You can combine optional output arguments when you set their enabling properties. Optional outputs must be listed in the same order as the order of the enabling properties. For example,

---

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

Filter input signal through MIMO multipath fading channel

Description

The `MIMOChannel` System object filters an input signal through a multiple-input multiple-output (MIMO) multipath fading channel. This object models both Rayleigh and Rician fading and employs the Kronecker model for modeling the spatial correlation between the links.

The fading processing per link is per the Methodology for Simulating Multipath Fading Channels section and assumes the same parameters for all $N_T \cdot N_R$ links of the MIMO channel. Each link comprises all multipaths for that link.

Construction

`H = comm.MIMOChannel` creates a multiple-input multiple-output (MIMO) frequency selective or frequency flat fading channel System object, `H`. This object filters a real or complex input signal through the multipath MIMO channel to obtain the channel impaired signal.

`H = comm.MIMOChannel(Name,Value)` creates a MIMO channel 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

Input signal sample rate (Hertz)

Specify the sample rate of the input signal in hertz as a double-precision, real, positive scalar. The default value of this property is `1` Hz.

PathDelays

Discrete path delay vector (seconds)

Specify the delays of the discrete paths in seconds as a double-precision, real, scalar or row vector. The default value of this property is 0. When you set `PathDelays` to a scalar, the MIMO channel is frequency flat. When you set `PathDelays` to a vector, the MIMO channel is frequency selective.

AveragePathGains

Average path gain vector (decibels)

Specify the average gains of the discrete paths in decibels as a double-precision, real, scalar or row vector. The default value of this property is `0`. `AveragePathGains` must have the same size as `PathDelays`.

MaximumDopplerShift

Maximum Doppler shift (Hertz)

Specify the maximum Doppler shift for all channel paths in hertz as a double precision, real, nonnegative scalar. The default value of this property is `0.001` Hz.

The Doppler shift applies to all the paths of the channel. When you set the `MaximumDopplerShift` to `0`, the channel remains static for the entire input. You can use the `reset` method to generate a new channel realization.

The `MaximumDopplerShift` must be smaller than `SampleRate`/10/f_c for each path, where f_c represents the cutoff frequency factor of the path. For a Doppler spectrum type other than Gaussian and BiGaussian, the value of f_c is `1`. For these two Doppler spectrum types, f_c is dependent on the Doppler spectrum object properties. Refer to the algorithm section of this page for more details about how f_c is defined.

DopplerSpectrum

Doppler spectrum object(s)

Specify the Doppler spectrum shape for all channel paths as a single object from the Doppler spectrum package or a row vector of such objects. The default value of this property is a Jakes Doppler spectrum object.

The maximum Doppler shift value necessary to specify the Doppler spectrum/spectra is given by the `MaximumDopplerShift` property. This property applies when the `MaximumDopplerShift` property value is greater than `0`.

If you assign a single Doppler spectrum object to `DopplerSpectrum`, all paths have the same specified Doppler spectrum. Select from the following:

- doppler.jakes
- doppler.flat
- doppler.rjakes(...)
- doppler.ajakes(...)
- doppler.rounded(...)
- doppler.bell(...)
- doppler.gaussian(...)
- doppler.bigaussian(...)

You can assign `DopplerSpectrum` a vector of Doppler spectrum objects, which can be chosen from any of those in the previous list. Each path has the Doppler spectrum specified by the corresponding Doppler spectrum object in the vector. In this case, the length of `DopplerSpectrum` must be equal to the length of `PathDelays`.

This object supports C code generation for the Jakes Doppler spectrum. You cannot explicitly specify the `DopplerSpectrum` property for code generation.

TransmitCorrelationMatrix

Transmit correlation matrix (or 3D array)

Specify the spatial correlation of the transmitter as a double-precision, real or complex, 2D matrix or 3D array. The default value of this property is `[1 0;0 1]`.

The first dimension of `TransmitCorrelationMatrix` determines the number of transmit antennas, N_t. This dimension must be a value between 1 and 8, inclusive.

If the channel is frequency flat, i.e., `PathDelays` is a scalar, `TransmitCorrelationMatrix` is a 2D Hermitian matrix of size N_t-by-N_t. The main diagonal elements must be all ones. The off-diagonal elements must be real or complex numbers with a magnitude smaller than or equal to one.

If the channel is frequency selective, i.e., `PathDelays` is a row vector of length N_p, you can specify `TransmitCorrelationMatrix` as an N_t-by-N_t matrix. In this case, each path has the same transmit spatial correlation matrix. Alternatively, you can specify the value as a 3D array of size N_t-by-N_t-by-N_p. In this case, each path can have its own transmit spatial correlation matrix.

ReceiveCorrelationMatrix

Receive correlation matrix (or 3D array)

Specify the spatial correlation of the receiver as a double-precision, real or complex, 2D matrix or 3D array. The default value of this property is `[1 0;0 1]`.

The first dimension of `ReceiveCorrelationMatrix` determines the number of receive antennas, N_r. This dimension must be a value between 1 and 8, inclusive.

If the channel is frequency flat, i.e., `PathDelays` is a scalar, `ReceiveCorrelationMatrix` is a 2D Hermitian matrix of size N_r-by-N_r. The main diagonal elements must be all ones. The off-diagonal elements must be real or complex numbers with a magnitude smaller than or equal to one.

If the channel is frequency selective, i.e., `PathDelays` is a row vector of length N_p, you can specify `ReceiveCorrelationMatrix` as an N_r-by-N_r matrix. In this case, each path has the same receive spatial correlation matrix. Alternatively, you can specify the value as a 3D array of size N_r-by-N_r-by-N_p. In this case, each path can have its own receive spatial correlation matrix.

AntennaSelection

Optional transmit and/or receive antenna selection

Specify the antenna selection scheme as one of `Off | Tx | Rx | Tx and Rx`. The default value of this property is `Off`.

`Tx` represents transmit antennas and `Rx` represents receive antennas. When you configure any antenna selection other than the default setting, the object requires one or more inputs to specify which antennas are selected for signal transmission.

FadingDistribution

Rayleigh or Rician fading

Specify the fading distribution of the channel as one of `Rayleigh| Rician`. The default value of this property is `Rayleigh`, i.e., the channel is Rayleigh fading.

KFactor

Rician K-factor scalar or vector (linear scale)

Specify the K factor of a Rician fading channel as a double-precision, real, positive scalar or nonnegative, nonzero row vector of the same length as `PathDelays`. This property applies when you set the `FadingDistribution` property to `Rician`. The default value of this property is `3`.

If `KFactor` is a scalar, the first discrete path is a Rician fading process with a Rician K factor of `KFactor`. The remaining discrete paths are independent Rayleigh fading processes. If `KFactor` is a row vector, the discrete path corresponding to a positive element of the `KFactor` vector is a Rician fading process with a Rician K factor specified by that element. The discrete path corresponding to a zero-valued element of the `KFactor` vector is a Rayleigh fading process.

DirectPathDopplerShift

Doppler shift(s) of line-of-sight component(s) (Hertz)

Specify the Doppler shifts for the line-of-sight components of a Rician fading channel in hertz as a double-precision, real scalar or row vector. The default value of this property is `0`. This property applies when you set the `FadingDistribution` property to `Rician`.

`DirectPathDopplerShift` must have the same size as `KFactor`. If `DirectPathDopplerShift` is a scalar, this value represents the line-of-sight component Doppler shift of the first discrete path. This path exhibits a Rician fading process. If `DirectPathDopplerShift` and `KFactor` are row vectors, the discrete path corresponding to a positive element of the `KFactor` vector is a Rician fading process. Its line-of-sight component Doppler shift is specified by the corresponding element of `DirectPathDopplerShift`.

DirectPathInitialPhase

Initial phase(s) of line-of-sight component(s) (radians)

Specify the initial phases of the line-of-sight components of a Rician fading channel in radians as a double precision, real scalar or row vector. The default value of this property is `0`. This property applies when you set the `FadingDistribution` property to `Rician`.

`DirectPathInitialPhase` must have the same size as `KFactor`. If `DirectPathInitialPhase` is a scalar, this value represents the line-of-sight component initial phase of the first discrete path. This path exhibits a Rician fading process. If `DirectPathInitialPhase` and `KFactor` are row vectors, the discrete path corresponding to a positive element of the `KFactor` vector is a Rician fading process. Its line-of-sight component initial phase is specified by the corresponding element of `DirectPathInitialPhase`.

RandomStream

Source of random number stream

Specify the source of random number stream as one of `Global stream | mt19937ar with seed`. The default value of this property is `Global stream`. If you set `RandomStream` to `Global stream`, the current global random number stream is used for normally distributed random number generation. In this case, the `reset` method only resets the filters. If you set `RandomStream` to `mt19937ar with seed`, the mt19937ar algorithm is used for normally distributed random number generation. In this case, the `reset` method not only resets the filters but also reinitializes the random number stream to the value of the `Seed` property.

Seed

Initial seed of mt19937ar random number stream

Specify the initial seed of a mt19937ar random number generator algorithm as a double precision, real, nonnegative integer scalar. The default value of this property is `73`. This property applies when you set the `RandomStream` property to `mt19937ar with seed`. The `Seed` reinitializes the mt19937ar random number stream in the `reset` method.

NormalizePathGains

Normalize path gains (logical)

Set this property to `true` to normalize the fading processes such that the total power of the path gains, averaged over time, is `0` dB. The default value of this property is `true`. When you set this property to `false`, there is no normalization on path gains. The average powers of the path gains are specified by the `AveragePathGains` property.

NormalizeChannelOutputs

Normalize channel outputs (logical)

Set this property to `true` to normalize the channel outputs by the number of receive antennas. The default value of this property is true. When you set this property to `false`, there is no normalization for channel outputs.

PathGainsOutputPort

Enable path gain output (logical)

Set this property to `true` to output the channel path gains of the underlying fading process. The default value of this property is `false`.

Methods

clone	Create object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the object
step	Filter input signal through MIMO multipath fading channel

Examples

Examine Spatial Correlation Characteristics of a 2-by-2 Rayleigh Fading Channel

Filter PSK modulated data through a 2-by-2 Rayleigh fading channel, and then examine the spatial correlation characteristics of the channel realization.

Create a PSK Modulator System object to modulate randomly generated data.

```
hMod = comm.PSKModulator;
modData = step(hMod, randi([0 hMod.ModulationOrder-1],2e6,1));
```

Split modulated data into two spatial streams.

```
channelInput = reshape(modData, [2, 1e6]).';
```

Create a 2-by-2 `MIMOChannel` System object with two discrete paths. Each path has different transmit and receive correlation matrices, specified by the `TransmitCorrelationMatrix` and `ReceiveCorrelationMatrix` properties.

```
hMIMOChan = comm.MIMOChannel(...
  'SampleRate',                1000,...
  'PathDelays',                [0, 1e-3],...
  'AveragePathGains',          [3, 5],...
  'MaximumDopplerShift',       5,...
  'TransmitCorrelationMatrix', cat(3, eye(2), [1 0.1;0.1 1]),...
  'ReceiveCorrelationMatrix',  cat(3, [1 0.2;0.2 1], eye(2)),...
  'RandomStream',              'mt19937ar with seed',...
  'Seed',                      33,...
  'NormalizePathGains',        false,...
  'PathGainsOutputPort',       true);
```

Filter the modulated data using `hMIMOChan` and use the `step` method to process data.

```
[channelOutput, pathGains] = step(hMIMOChan, channelInput);
```

The transmit spatial correlation for the first discrete path at the first receive antenna is specified as an identity matrix in the `TransmitCorrelationMatrix` property. Confirm that the channel output `pathGains` exhibits the same statistical characteristics using the `corrcoef` function.

```
disp('Tx spatial correlation, first path, first Rx:');
disp(corrcoef(squeeze(pathGains(:,1,:,1))));
```

The transmit spatial correlation for the second discrete path at the second receive antenna is specified as `[1 0.1;0.1 1]` in the

TransmitCorrelationMatrix property. Confirm that the channel output pathGains exhibits the same statistical characteristics.

```
disp('Tx spatial correlation, second path, second Rx:');
disp(corrcoef(squeeze(pathGains(:,2,:,2))));
```

The receive spatial correlation for the first discrete path at the second transmit antenna is specified as [1 0.2;0.2 1] in the ReceiveCorrelationMatrix property. Confirm that the channel output pathGains exhibits the same statistical characteristics.

```
disp('Rx spatial correlation, first path, second Tx:');
disp(corrcoef(squeeze(pathGains(:,1,2,:))));
```

The receive spatial correlation for the second discrete path at the first transmit antenna is specified as an identity matrix in the ReceiveCorrelationMatrix property. Confirm that the channel output pathGains exhibits the same statistical characteristics.

```
disp('Rx spatial correlation, second path, first Tx:');
disp(corrcoef(squeeze(pathGains(:,2,1,:))));
```

Now enable transmit and receive antenna selection for the System object hMIMOChan. The input frame size is shortened to 100.

```
release(hMIMOChan);
hMIMOChan.AntennaSelection = 'Tx and Rx';
modData = step(hMod,randi([0 hMod.ModulationOrder-1],1e2,1));
```

Select the first transmit and second receive antennas.

```
[channelOutput, pathGains] = step(hMIMOChan, modData, [1 0], [0 1]);
```

Confirm that the path gains MATLAB returns have NaN values for the unselected transmit-receive antenna pairs.

```
disp('Return 1 if the path gains for the second transmit antenna are NaN
disp(isequal(isnan(squeeze(pathGains(:,:,2,:))), ones(1e2, 2, 2)));
disp('Return 1 if the path gains for the first receive antenna are NaN:'
```

```
disp(isequal(isnan(squeeze(pathGains(:,:,:,1))), ones(1e2, 2, 2)));
```

Algorithms

The fading processing per link is per the Methodology for Simulating Multipath Fading Channels section and assumes the same parameters for all $N_T \cdot N_R$ links of the MIMO channel. Each link comprises all multipaths for that link.

The Kronecker Model

The Kronecker model assumes that the spatial correlations at the transmit and receive sides are separable. Equivalently, the direction of departure (DoD) and directions of arrival (DoA) spectra are assumed to be separable. The full correlation matrix can then be obtained as:

$$R_H = E\left[R_t \otimes R_r\right]$$

where:

The $\otimes$ symbol represents the Kronecker product.

R_t represents the correlation matrix at the transmit side, i.e. $R_t = E\left[H^H H\right]$, of size N_t-by-N_t.

R_r represents the correlation matrix at the receive side, i.e. $R_r = E\left[H H^H\right]$, of size N_r-by-N_r.

You can obtain a realization of the MIMO channel matrix as:

$$H = R_r^{\frac{1}{2}} A R_t^{\frac{1}{2}}$$

where: A = unvec(a) is an N_r-by-N_t matrix of i.i.d. complex Gaussian variables with zero mean and unit variance.

Note that the $\otimes$ symbol represents the Kronecker product.

Cutoff Frequency Factor

The following information explains how this objects determines the cutoff frequency factor, f_c:

- For any Doppler spectrum type, other than Gaussian and BiGaussian, f_c equals `1`.
- For a Gaussian Doppler spectrum type, f_c equals the Doppler spectrum object `SigmaGaussian` property value times sqrt(2·log(2)).
- For a BiGaussian Doppler spectrum type:
  - If the Doppler spectrum object `GainGaussian1` (`GainGaussian2`) and `CenterFreqGaussian2` (`CenterFreqGaussian1`) property values are both 0, then f_c equals the `SigmaGaussian2` (`SigmaGaussian1`) property value times sqrt(2·log(2)).
  - If the `CenterFreqGaussian1` and `CenterFreqGaussian2` property values are both `0` and the `SigmaGaussian1` and `SigmaGaussian2` property values are the same, then f_c is also equal to the `SigmaGaussian2` property value times sqrt(2·log(2)).
  - In all other cases, f_c equals `1`.

Antenna Selection

When the object is in antenna selection mode, it uses the following algorithms to process an input signal:

- The random path gains are always generated and keep evolving for each link, no matter whether the links is being selected or not. The path gain values for the non-selected links are marked as `NaN` in the path gain output.
- The spatial correlation only applies to the selected transmit and/or receive antennas, and the correlation coefficients are the corresponding entries in the transmit and/or receive correlation matrices. In other words, the spatial correlation matrix for the selected transmit/receive antennas is a submatrix of the

`TransmitCorrelationMatrix/ReceiveCorrelationMatrix` property value.

- The input filtering through the path gains only happens to the selected links. Channel output normalization happens over the number of selected receive antennas.

Selected Bibliography

[1] Oestges, C., and B. Clerckx. *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*, Academic Press, 2007.

[2] Correira, L. M. *Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G*, Academic Press, 2006.

[3] Kermoal, J. P., L. Schumacher, K. I. Pedersen, P. E. Mogensen, and F. Frederiksen. “A stochastic MIMO radio channel model with experimental validation." *IEEE Journal on Selected Areas of Communications*. Vol. 20, Number 6, 2002, pp. 1211–1226.

[4] Jeruchim, M., P. Balaban, and K. S. Shanmugan. *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

See Also

`comm.LTEMIMOChannel`

Purpose Create `MIMOChannel` object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `MIMOChannel` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(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 from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

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

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `MIMOChannel` 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.

comm.MIMOChannel.release

Purpose

Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description

`release(H)` releases system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the `MIMOChannel` object

Syntax

```
reset(H)
```

Description `reset(H)` resets the states of the `MIMOChannel` object, `H`.

If you set the `RandomStream` property of `H` to `Global stream`, the `reset` method only resets the filters. If you set `RandomStream` to `mt19937ar with seed`, the `reset` method not only resets the filters but also reinitializes the random number stream to the value of the `Seed` property.

Purpose

Filter input signal through MIMO multipath fading channel

Syntax

```
Y = step(H,X)
Y = step(H,X,SELTX)
Y = step(H,X,SELRX)
Y = step(H,X,SELTX,SELRX)
[Y,PATHGAINS] = step(H,X)
[Y,PATHGAINS] = step(H,X,SELTX/SELRX)
step(H,X,SELTX,SELRX)
```

Description

`Y = step(H,X)` filters input signal `X` through a MIMO fading channel and returns the result in `Y`. The input `X` can be a double-precision data type scalar, vector, or 2-D matrix with real or complex values. `X` is of size N_s-by-N_t, where N_s represents the number of samples and N_t represents the number of transmit antennas that is determined by the `TransmitCorrelationMatrix` property value of `H`. `Y` is the output signal of size N_s-by-N_r, where N_r represents the number of receive antennas that is determined by the `ReceiveCorrelationMatrix` property value of `H`. `Y` is of double-precision data type with complex values.

`Y = step(H,X,SELTX)` turns on selected transmit antennas for `X` transmission. This syntax applies when you set the `AntennaSelection` property of `H` to `Tx`. `SELTX` is a numeric type binary-valued 1-by-N_t row vector. In this row vector, the ones indicate the selected transmit antennas. `X` is size N_s-by-N_{st}, where N_{st} represents the number of selected transmit antennas, i.e., the number of ones in `SELTX`. `Y` is size N_s-by-N_r.

`Y = step(H,X,SELRX)` turns on selected receive antennas for `X` transmission. This syntax applies when you set the `AntennaSelection` property of `H` to `Rx`. `SELRX` is a numeric type binary-valued 1-by-N_r row vector, in which the ones indicate the selected receive antennas. `X` is of size N_s-by-N_t. `Y` is of size N_s-by-N_{sr}, where N_{sr} represents the number of selected receive antennas, i.e., the number of ones in `SELRX`.

`Y = step(H,X,SELTX,SELRX)` turns on selected transmit and receive antennas for `X` transmission. This syntax applies when you set the `AntennaSelection` property of `H` to `Tx and Rx`. `X` is of size N_s-by-N_{st}, and `Y` is of size N_s-by-N_{sr}.

`[Y,PATHGAINS] = step(H,X)` returns the MIMO channel path gains of the underlying fading process in PATHGAINS. This syntax applies when you set the `PathGainsOutputPort` property of `H` to true. PATHGAINS is of size N_s-by-N_p-by-N_t-by-N_r, where N_p represents the number of paths, i.e., the length of the `PathDelays` property value of `H`. PATHGAINS is of double-precision data type with complex values.

`[Y,PATHGAINS] = step(H,X,SELTX/SELRX)` or `step(H,X,SELTX,SELRX)` returns the MIMO channel path gains for antenna selection schemes. PATHGAINS is still of size N_s-by-N_p-by-N_t-by-N_r with NaN values for the unselected transmit-receive antenna pairs.

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

Equalize using maximum likelihood sequence estimation

Description

The `MLSEEqualizer` object uses the Viterbi algorithm to equalize a linearly modulated signal through a dispersive channel. The object processes input frames and outputs the maximum likelihood sequence estimate (MLSE) of the signal. This processing uses an estimate of the channel modeled as a finite impulse response (FIR) filter.

Construction

`H = comm.MLSEEqualizer` creates a maximum likelihood sequence estimation equalizer (MLSEE) System object, `H`. This object uses the Viterbi algorithm and a channel estimate to equalize a linearly modulated signal that has been transmitted through a dispersive channel.

`H = comm.MLSEEqualizer(Name,Value)` creates an MLSEE 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`).

`H = comm.MLSEEqualizer(CHANNEL,Name,Value)` creates an MLSEE object, `H`. This object has the `Channel` property set to `CHANNEL`, and the other specified properties set to the specified values.

Properties

ChannelSource

Source of channel coefficients

Specify the source of the channel coefficients as one of `Input port | Property`. The default is `Property`.

Channel

Channel coefficients

Specify the channel as a numeric, column vector containing the coefficients of an FIR filter. The default is `[1;0.7;0.5;0.3]`. The length of this vector determines the memory length of the channel. This must be a multiple of the samples per symbol, that you specify in the `SamplesPerSymbol` property. This property applies when you set the `ChannelSource` property to `Property`.

Constellation

Input signal constellation

Specify the constellation of the input modulated signal as a complex vector. The default is `[1+1i -1+1i -1-1i 1-1i]`.

TracebackDepth

Traceback depth of Viterbi algorithm

Specify the number of trellis branches (the number of symbols), the Viterbi algorithm uses to construct each traceback path. The default is `21`. The traceback depth influences the decoding accuracy and delay. The decoding delay represents the number of zero symbols that precede the first decoded symbol in the output. When you set the `TerminationMethod` property to `Continuous`, the decoding delay equals the number of zero symbols of this property. When you set the `TerminationMethod` property to `Truncated`, there is no output delay.

TerminationMethod

Termination method of Viterbi algorithm

Specify the termination method of the Viterbi algorithm as one of `Continuous | Truncated`. The default is `Truncated`. When you set this property to `Continuous`, the object initializes the Viterbi algorithm metrics of all the states to `0` in the first call to the `step` method. Then, the object saves its internal state metric at the end of each frame, for use with the next frame. When you set this property to `Truncated`, the object resets at every frame. The Viterbi algorithm processes each frame of data independently, resetting the state metric at the end of each frame. The traceback path always starts at the state with the minimum metric. The initialization of the state metrics depends on whether you specify a preamble or postamble. If you set the `PreambleSource` property to `None`, the object initializes the metrics of all the states to `0` at the beginning of each data frame. If you set the `PreambleSource` property to `Property`, the object uses the preamble that you specify at the `Preamble` property, to initialize the state metrics at the beginning of each data frame. When you specify a preamble, the traceback path ends at one of the states represented by that preamble. If you set the `PostambleSource` property to `None`, the traceback path starts at the state with the smallest metric. If you set the `PostambleSource` property to `Property`, the traceback path begins at the state represented by the postamble that you specify at the `Postamble` property. If the postamble does not decode to a unique state, the decoder identifies the smallest of all possible decoded states that are represented by the postamble. The decoder then begins traceback decoding at that state. When you set this property to `Truncated`, the `step` method input data signal must contain at least `TracebackDepth` symbols, not including an optional preamble.

ResetInputPort

Enable equalizer reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this input is a nonzero, double-precision or logical scalar value, the object resets the states of the equalizer. This property applies when you set the `TerminationMethod` property to `Continuous`.

PreambleSource

Source of preamble

Specify the source of the preamble that is expected to precede the input signal. Choose from `None | Property`. The default is `None`. Set this property to `Property` to specify a preamble using the `Preamble` property. This property applies when you set the `TerminationMethod` property to `Truncated`.

Preamble

Preamble that precedes input signals

Specify a preamble that is expected to precede the data in the input signal as an integer, row vector. The default is `[0 3 2 1]`. The values of the preamble should be between `0` and *M*-`1`, where M is the length of the signal constellation that you specify in the `Constellation` property. An integer value of k–1 in the vector corresponds to the k-th entry in the vector stored in the `Constellation` property. This property applies when you set the `TerminationMethod` property to `Truncated` and the `PreambleSource` property to `Property`.

PostambleSource

Source of postamble

Specify the source of the postamble that is expected to follow the input signal. Choose from `None | Property`. The default is `None`. Set this property to `Property` to specify a postamble in the `Postamble` property. This property applies when you set the `TerminationMethod` property to `Truncated`.

Postamble

Postamble that follows input signals

Specify a postamble that is expected to follow the data in the input signal as an integer row vector. The default is `[0 2 3 1]`. The values of the postamble should be between `0` and M–1. In this case, M indicates the length of the `Constellation` property. An integer value of k–1 in the vector corresponds to the k-th entry in the vector specified in the `Constellation` property. This property applies when you set the `TerminationMethod` property to `Truncated` and the `PostambleSource` property to `Property`. The default is `[0 2 3 1]`.

SamplesPerSymbol

Number of samples per symbol

Specify the number of samples per symbol in the input signal as an integer scalar value. The default is `1`.

Methods

clone	Create MLSEE object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of MLSEE object
step	Equalize using maximum likelihood sequence estimation

Examples

Equalize a QPSK signal transmitted through a dispersive channel.

```
   hMod = comm.QPSKModulator(0,'SymbolMapping','Binary');
   hDemod = comm.QPSKDemodulator(0,'SymbolMapping','Binary');
% Channel coefficients
   chCoeffs = [.986; .845; .237; .12345+.31i];
   hMLSEE = comm.MLSEEqualizer('TracebackDepth',10,...
                    'Channel',chCoeffs, 'Constellation',[1 1i -1 -1i]);
% Create an error rate calculator
   hError = comm.ErrorRate;
   for n = 1:50
     data= randi([0 3],100,1);
     modSignal = step(hMod, data);

% Introduce channel distortion.
     chanOutput = filter(chCoeffs,1,modSignal);

% Equalize the channel output and demodulate
     eqSignal = step(hMLSEE,chanOutput);
```

```
      demodData = step(hDemod,eqSignal);
 % Compute BER
      errorStats = step(hError, data, demodData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the MLSE Equalizer block reference page. The object properties correspond to the block parameters.

See Also

`comm.ViterbiDecoder`

comm.MLSEEqualizer.clone

Purpose

Create MLSEE object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `MLSEEqualizer` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.MLSEEqualizer.getNumOutputs

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `MLSEEqualizer` 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.

comm.MLSEEqualizer.release

Purpose

Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 MLSEE object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `MLSEEqualizer` object, `H`.

Purpose

Equalize using maximum likelihood sequence estimation

Syntax

```
Y = step(H,X)
Y = step(H,X,CHANNEL)
Y = step(H,X,RESET)
Y = step(H,X,CHANNEL,RESET)
```

Description

`Y = step(H,X)` equalizes the linearly modulated data input, `X`, using the Viterbi algorithm. The step method outputs `Y`, the maximum likelihood sequence estimate of the signal. Input `X` must be a column vector of data type double or single.

`Y = step(H,X,CHANNEL)` uses `CHANNEL` as the channel coefficients when you set the `ChannelSource` property to 'Input port'. The channel coefficients input, `CHANNEL`, must be a numeric, column vector containing the coefficients of an FIR filter in descending order of powers of z. The length of this vector is the channel memory, which must be a multiple of the samples per input symbol specified in the `SamplesPerSymbol` property.

`Y = step(H,X,RESET)` uses `RESET` as the reset signal when you set the `TerminationMethod` property to 'Continuous' and the `ResetInputPort` property to true. The object resets when `RESET` has a non-zero value. `RESET` must be a double precision or logical scalar. You can combine optional input arguments when you set their enabling properties. Optional inputs must be listed in the same order as the order of the enabling properties. For example,

```
Y = step(H,X,CHANNEL,RESET)
```

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

Demodulate using MSK method and the Viterbi algorithm

Description

The `MSKDemodulator` object demodulates a signal that was modulated using the minimum shift keying method. The input is a baseband representation of the modulated signal. The initial phase offset property sets the initial phase of the modulated waveform.

Construction

`H = comm.MSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input minimum shift keying (MSK) modulated data using the Viterbi algorithm.

`H = comm.MSKDemodulator(Name,Value)` creates an MSK demodulator 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

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer values. The default is `false`.

When you set this property to `false`, the `step` method outputs a column vector with a length equal to N/`SamplesPerSymbol`. N represents the length of the input signal, which is the number of input baseband modulated symbols. The elements of the output vector are `-1` or `1`.

When you set the `BitOutput` property to `true`, the `step` method outputs a binary column vector with a length equal to N/`SamplesPerSymbol`. The vector elements are bit values of `0` or `1`.

InitialPhaseOffset

Initial phase offset

Specify the initial phase offset of the input modulated waveform in radians as a real, numeric scalar value. The default is `0`.

SamplesPerSymbol

Number of samples per input symbol

Specify the expected number of samples per input symbol as a positive, integer scalar value. The default is `8`.

TracebackDepth

Traceback depth for Viterbi algorithm

Specify the number of trellis branches that the Viterbi algorithm uses to construct each traceback path as a positive, integer scalar value. The default is `16`. The value of this property is also the output delay This value indicates number of zero symbols that precede the first meaningful demodulated symbol in the output.

OutputDataType

Data type of output

Specify the output data type as one of `int8 | int16 | int32 | double`, when you set the `BitOutput` property to false. The default is `double`.

When you set the `BitOutput` property to true, specify the output data type as one of `logical | double`.

Methods

clone	Create MSK demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes

reset	Reset states of the MSK demodulator object
step	Demodulate using MSK method and the Viterbi algorithm

Examples

Modulate and demodulate a signal using MSK modulation with bit inputs and an initial phase offset of pi/4.

```
    hMod = comm.MSKModulator('BitInput', true, 'InitialPhaseOffset', pi/4
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.MSKDemodulator('BitOutput', true, ...
                    'InitialPhaseOffset', pi/4);
% Create an error rate calculator, account for the delay caused by the Vi
    hError = comm.ErrorRate('ReceiveDelay', hDemod.TracebackDepth);
    for counter = 1:100
      % Transmit 100 3-bit words
      data = randi([0 1],300,1);
      modSignal = step(hMod, data);
      noisySignal = step(hAWGN, modSignal);
      receivedData = step(hDemod, noisySignal);
      errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the MSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

comm.MSKModulator | comm.CPMModulator | comm.CPMDemodulator

Purpose

Create MSK demodulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `MSKDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.MSKDemodulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.MSKDemodulator.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `MSKDemodulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

comm.MSKDemodulator.reset

Purpose Reset states of the MSK demodulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `MSKDemodulator` object, `H`.

Purpose

Demodulate using MSK method and the Viterbi algorithm

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` demodulates input data, `X`, with the MSK demodulator System object, `H`, and returns `Y`. `X` must be a double or single precision column vector with a length equal to an integer multiple of the number of samples per symbol you specify in the `SamplesPerSymbol` 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.

---

comm.MSKModulator

Purpose

Modulate using MSK method

Description

The `MSKModulator` object modulates using the minimum shift keying method. The output is a baseband representation of the modulated signal. The initial phase offset property sets the initial phase of the output waveform, measured in radians.

Construction

`H = comm.MSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the minimum shift keying (MSK) modulation method.

`H = comm.MSKModulator(Name,Value)` creates an MSK modulator 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

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`.

When you set the `BitInput` property to `false`, the `step` method input must be a column vector with a double-precision or signed integer data type and of values equal to `-1` or `1`.

When you set the `BitInput` property to `true`, the `step` method input requires double-precision or logical data type column vector of 0s and 1s.

InitialPhaseOffset

Initial phase offset

Specify the initial phase of the modulated waveform in radians as a real, numeric scalar value. The default is `0`.

SamplesPerSymbol

Number of samples per output symbol

Specify the upsampling factor at the output as a real, positive, integer scalar value. The default is 8. The upsampling factor indicates the number of output samples that the step method produces for each input sample.

OutputDataType

Data type of output

Specify output data type as one of `double | single`. The default is `double`.

Methods

clone	Create MSK modulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the MSK modulator object
step	Modulate using MSK method

Examples

Modulate and demodulate a signal using MSK modulation with bit inputs and an initial phase offset of pi/4

```
hMod = comm.MSKModulator('BitInput', true, ...
                'InitialPhaseOffset', pi/4);
hAWGN = comm.AWGNChannel('NoiseMethod', ...
                'Signal to noise ratio (SNR)','SNR',0);
hDemod = comm.MSKDemodulator('BitOutput', true, ...
```

```
                   'InitialPhaseOffset', pi/4);
% Create an error rate calculator, account for the delay caused by the V
   hError = comm.ErrorRate('ReceiveDelay', hDemod.TracebackDepth);
   for counter = 1:100
     % Transmit 100 3-bit words
     data = randi([0 1],300,1);
     modSignal = step(hMod, data);
     noisySignal = step(hAWGN, modSignal);
     receivedData = step(hDemod, noisySignal);
     errorStats = step(hError, data, receivedData);
   end
   fprintf('Error rate = %f\nNumber of errors = %d\n', ...
     errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the MSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.MSKDemodulator` | `comm.CPMModulator` | `comm.CPMDemodulator`

Purpose

Create MSK modulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `MSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.MSKModulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.MSKModulator.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `MSKModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.MSKModulator.reset

Purpose Reset states of the MSK modulator object

Syntax

```
reset(H)
```

Description `reset(H)` resets the states of the `MSKModulator` object, `H`.

Purpose

Modulate using MSK method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` modulates input data, `X`, with the MSK modulator object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be a double precision, signed integer, or logical column vector. The length of output vector, `Y`, is equal to the number of input samples times the number of samples per symbol you specify in the `SamplesPerSymbol` 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

Recover symbol timing phase using fourth-order nonlinearity method

Description

The `MSKTimingSynchronizer` object recovers the symbol timing phase of the input signal using a fourth-order nonlinearity method. This object implements a general non-data-aided feedback method that is independent of carrier phase recovery. This method requires prior compensation for the carrier frequency offset. This object is suitable for systems that use baseband minimum shift keying (MSK) modulation.

Construction

`H = comm.MSKTimingSynchronizer` creates a timing phase synchronizer System object, `H`. This object recovers the symbol timing phase of the input signal using a fourth-order nonlinearity method.

`H = comm.MSKTimingSynchronizer(Name,Value)` creates an MSK timing synchronizer 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

SamplesPerSymbol

Number of samples representing each symbol

Specify the number of samples that represent each symbol in the input signal as an integer-valued scalar greater than `1`. The default is `4`.

ErrorUpdateGain

Error update step size

Specify the step size for updating successive timing phase estimates as a positive, real scalar value. The default is `0.05`. Typically, this number is less than 1/`SamplesPerSymbol`, which corresponds to a slowly varying timing phase. This property is tunable.

ResetInputPort

Enable synchronization reset input

Set this property to `true` to enable resetting the timing phase recovery process based on an input argument value. The default is `false`.

When you set this property to `true`, you must specify a reset input value to the `step` method.

When the reset input is a nonzero value, the object restarts the timing phase recovery process. When you set this property to `false`, the object does not restart.

ResetCondition

Condition for timing phase recovery reset

Specify the conditions to reset the timing phase recovery process as one of `Never | Every frame`. The default is `Never`.

When you set this property to `Never`, the phase recovery process never restarts. The object operates continuously, retaining information from one symbol to the next.

When you set this property to `Every frame`, the timing phase recovery restarts at the start of each frame of data. Thus, each time the object calls the step method. This property applies when you set the `ResetInputPort` property to `false`.

Methods

clone	Create MSK timing phase synchronizer object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
reset	Reset states of MSK timing phase synchronizer object
step	Recover symbol timing phase using fourth-order nonlinearity method

Examples

Recover timing phase of an MSK signal.

```
% Create System objects
   hMod = comm.MSKModulator('BitInput', true,...
                   'SamplesPerSymbol', 14);
   timingOffset = 0.2; % Actual timing offset
   hDelay = dsp.VariableFractionalDelay;
   hSync = comm.MSKTimingSynchronizer('SamplesPerSymbol', 14, ...
                   'ErrorUpdateGain', 0.05);
   phEst = zeros(1, 10);
   for i = 1:51
     data = randi([0 1], 100, 1); % generate data
     modData = step(hMod, data);  % modulate data

% data impaired by timing offset error
     impairedData = step(hDelay, modData, timingOffset*14);
     % perform timing phase recovery
     [y, phase] = step(hSync, impairedData);
     phEst(i) = phase(1)/14;
   end

   figure, plot(0.2*ones(1, 50));
   hold on; ylim([0 0.4])
   plot(phEst, 'r'); legend( 'original', 'estimated')
   title('Original and Estimated timing phases');
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the MSK-Type Signal Timing Recovery block reference page. The object properties correspond to the block parameters, except:

- The object corresponds to the MSK-Type Signal Timing Recovery block with the **Modulation type** parameter set to `MSK`.
- The **Reset** parameter corresponds to the `ResetInputPort` and `ResetCondition` properties.

See Also

`comm.EarlyLateGateTimingSynchronizer` | `comm.MuellerMullerTimingSynchronizer`

Purpose

Create MSK timing phase synchronizer object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `MSKTimingSynchronizer` 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)` method returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`.

comm.MSKTimingSynchronizer.getNumOutputs

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `MSKTimingSynchronizer` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 MSK timing phase synchronizer object

Syntax

```
reset(H)
```

Description `reset(H)` resets the states of `MSKTimingSynchronizer` object, `H`.

comm.MSKTimingSynchronizer.step

Purpose

Recover symbol timing phase using fourth-order nonlinearity method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Description

`[Y,PHASE] = step(H,X)` recovers the timing phase and returns the time-synchronized signal, `Y`, and the estimated timing phase, `PHASE`, for input signal `X`. `X` must be a double or single precision complex column vector.

`[Y,PHASE] = step(H,X,R)` restarts the timing phase recovery process when you input a reset signal, `R`, that is non-zero. `R` must be a logical or double scalar. This syntax applies 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.

Purpose

Recover symbol timing phase using Mueller-Muller method

Description

The `MuellerMullerTimingSynchronizer` object recovers the symbol timing phase of the input signal using the Mueller-Muller method. This object implements a decision-directed, data-aided feedback method that requires prior recovery of the carrier phase.

Construction

`H = comm.MuellerMullerTimingSynchronizer` creates a timing synchronizer System object, `H`. This object recovers the symbol timing phase of the input signal using the Mueller-Muller method.

`H = comm.MuellerMullerTimingSynchronizer(Name,Value)` creates a Mueller-Muller timing recovery 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

SamplesPerSymbol

Number of samples representing each symbol

Specify the number of samples that represent each symbol in the input signal as an integer-valued scalar greater than `1`. The default is `4`.

ErrorUpdateGain

Error update step size

Specify the step size for updating successive timing phase estimates as a positive real scalar value. The default is `0.05`. Typically, this number is less than 1/`SamplesPerSymbol`, which corresponds to a slowly varying timing phase. This property is tunable.

ResetInputPort

Enable synchronization reset input

Set this property to `true` to enable resetting the timing phase recovery process based on an input argument value. The default is `false`. When you set this property to `true`, you must specify a reset input value to the `step` method. When the reset input is a nonzero value, the object restarts the timing phase recovery process. When you set this property to `false`, the object does not restart.

ResetCondition

Condition for timing phase recovery reset

Specify the conditions to reset the timing phase recovery process as `Never | Every frame`. The default is `Never`. When you set this property to `Never`, the phase recovery process never restarts. The object operates continuously, retaining information from one symbol to the next. When you set this property to `Every frame`, the timing phase recovery restarts at the start of each frame of data. Thus, restart occurs each time the object calls the step method. This property applies when you set the `ResetInputPort` property to `false`.

Methods

clone	Create Mueller-Muller timing phase synchronizer object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of Mueller-Muller timing phase synchronizer
step	Recover symbol timing phase using Mueller-Muller method

Examples

Recover timing phase using the Mueller-Muller method.

```
% Initialize some data
    L = 16; M = 2; numSymb = 100; snrdB = 30;
    R = 25; rollOff = 0.75; filtDelay = 3; g = 0.07; delay = 6.6498;
    % Design raised cosine filters
    txFiltSpec = fdesign.pulseshaping(L, 'Square root raised cosine',
                    'Nsym,Beta', 2*filtDelay, rollOff);
    txFilterDesign = design(txFiltSpec);
    txFilterDesign.Numerator = sqrt(L)*txFilterDesign.Numerator;

% Create System objects
    hMod = comm.DPSKModulator(M, 'PhaseRotation', 0);
    hTxFilter = dsp.FIRInterpolator(L, txFilterDesign.Numerator);
    hDelay = dsp.VariableFractionalDelay('MaximumDelay', L);
    hChan = comm.AWGNChannel(...
                    'NoiseMethod',  'Signal to noise ratio (SNR)', ..
                    'SNR', snrdB, 'SignalPower', 1/L);
    hRxFilter = dsp.DigitalFilter('TransferFunction', 'FIR (all zeros)
                    'Numerator', txFilterDesign.Numerator);
    hSync = comm.MuellerMullerTimingSynchronizer('SamplesPerSymbol', L
                    'ErrorUpdateGain', g);

% Generate random data
    data = randi([0 M-1], numSymb, 1);

% Modulate and filter transmitter data
    modData = step(hMod, data);
    filterData = step(hTxFilter, modData);

% Introduce a random delay.
    delayedData = step(hDelay, filterData, delay);

% Add noise
    chData = step(hChan, delayedData);

% Filter the receiver data
```

```
    rxData = step(hRxFilter, chData);

% Estimate the delay from the received signal
    [~, phase] = step(hSync, rxData);
    fprintf(1, 'Actual Timing Delay: %f\n', delay);
    fprintf(1, 'Estimated Timing Delay: %f\n', phase(end));
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Mueller-Muller Timing Recovery block reference page. The object properties correspond to the block parameters, except:

> The **Reset** parameter corresponds to the `ResetInputPort` and `ResetCondition` properties.

See Also

`comm.EarlyLateGateTimingSynchronizer` | `comm.GMSKTimingSynchronizer`

Purpose

Create Mueller-Muller timing phase synchronizer object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `MuellerMullerTimingSynchronizer` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.MuellerMullerTimingSynchronizer.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` method returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`.

comm.MuellerMullerTimingSynchronizer.getNumOutputs

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.MuellerMullerTimingSynchronizer.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `MuellerMullerTimingSynchronizer` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.MuellerMullerTimingSynchronizer.reset

Purpose Reset states of Mueller-Muller timing phase synchronizer

Syntax `reset(H)`

Description `reset(H)` resets the states of the `MuellerMullerTimingSynchronizer` object, `H`.

Purpose

Recover symbol timing phase using Mueller-Muller method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Description

`[Y,PHASE] = step(H,X)` performs timing phase recovery and returns the time-synchronized signal, `Y`, and the estimated timing phase, `PHASE`, for input signal `X`. The input `X` must be a double or single precision complex column vector. The length of `X` is N*K, where N is the value you specify in the property SamplesPerSymbol and K is the number of symbols. The output, `Y`, is the signal value for each symbol, which you use to make symbol decisions. `Y` is a column vector of length K with the same data type as `X`.

`[Y,PHASE] = step(H,X,R)` restarts the timing phase recovery process when you input a reset signal, `R`, that is non-zero. `R` must be a logical or double scalar. This syntax applies 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.

---

comm.MultiplexedDeinterleaver

Purpose

Deinterleave input symbols using set of shift registers with specified delays

Description

The `MultiplexedDeinterleaver` object restores the original ordering of a sequence that was interleaved using the General Multiplexed Interleaver object.

Construction

`H = comm.MultiplexedDeinterleaver` creates a multiplexed deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using the multiplexed interleaver System object.

`H = comm.MultiplexedDeinterleaver(Name,Value)` creates a multiplexed deinterleaver 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

Delay

Interleaver delay

Specify the lengths of the shift registers as an integer column vector. The default is `[2;0;1;3;10]`.

InitialConditions

Initial conditions of shift registers

Specify the initial values in each shift register as a numeric scalar value or a column vector. The default is `0`. When you set this property to a column vector, the vector length must equal the value of the `Delay` property. This vector contains initial conditions, where the *i*-th initial condition is stored in the *i*th shift register.

Methods

clone	Create multiplexed deinterleaver object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the multiplexed deinterleaver object
step	Deinterleave input symbols using a set of shift registers with specified delays

Examples

Interleave a sequence, and then restore it.

```
hInt = comm.MultiplexedInterleaver('Delay', [1 0 2]');
hDeInt = comm.MultiplexedDeinterleaver('Delay', [1 0 2]');
data = (1:20)';
intData = step(hInt, data);
deIntData = step(hDeInt, intData);

% compare the original sequence, interleaved sequence and restored
display([data intData deIntData]);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General Multiplexed Deinterleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.MultiplexedInterleaver` | `comm.ConvolutionalDeinterleaver`

comm.MultiplexedDeinterleaver.clone

Purpose

Create multiplexed deinterleaver object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `MultiplexedDeinterleaver` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.MultiplexedDeinterleaver.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `MultiplexedDeinterleaver` 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.

comm.MultiplexedDeinterleaver.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the multiplexed deinterleaver object

Syntax

```
reset(H)
```

Description `reset(H)` resets the states of the `MultiplexedDeinterleaver` object, `H`.

comm.MultiplexedDeinterleaver.step

Purpose

Deinterleave input symbols using a set of shift registers with specified delays

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a multiplexed interleaver and returns `Y`. The input `X` must be a column vector. The data type for `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The multiplexed deinterleaver object uses N shift registers, where N is the number of elements in the vector specified by the `Delay` property. When a new input symbol enters the deinterleaver, a commutator switches to a new register. The new symbol shifts in while the oldest symbol in that register is shifted out. When the commutator reaches the Nth register, upon the next new input, it returns to the first register. The multiplexed deinterleaver associated with a multiplexed interleaver has the same number of registers as the interleaver. The delay in a particular deinterleaver register depends on the largest interleaver delay minus the interleaver delay for the given register.

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

Permute input symbols using set of shift registers with specified delays

Description

The `MultiplexedInterleaver` object permutes the symbols in the input signal. Internally, the object uses a set of shift registers, each with its own delay value.

Construction

`H = comm.MultiplexedInterleaver` creates a multiplexed interleaver System object, `H`. This object permutes the symbols in the input signal using a set of shift registers with specified delays.

`H = comm.MultiplexedInterleaver(Name,Value)` creates a multiplexed interleaver 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

Delay

Interleaver delay

Specify the lengths of the shift registers as an integer column vector. The default is `[2;0;1;3;10]`.

InitialConditions

Initial conditions of shift registers

Specify the initial values in each shift register as a numeric scalar value or a column vector. The default is `0`. When you set this property to a column vector, the length must equal the value of the `Delay` property. This vector contains initial conditions, where the *i*-th initial condition is stored in the *i*-th shift register.

Methods

clone	Create multiplexed interleaver object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the multiplexed interleaver object
step	Permute input symbols using a set of shift registers with specified delays

Examples

Interleave a sequence, and then restore it.

```
hInt = comm.MultiplexedInterleaver('Delay', [1 0 2]');
hDeInt = comm.MultiplexedDeinterleaver('Delay', [1 0 2]');
data = (1:20)';
intData = step(hInt, data);
deIntData = step(hDeInt, intData);
% compare the original sequence, interleaved sequence, and restored
% sequence
[data, intData, deIntData]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General Multiplexed Interleaver block reference page. The object properties correspond to the block parameters.

See Also

`comm.MultiplexedDeinterleaver` | `comm.ConvolutionalInterleaver`

Purpose Create multiplexed interleaver object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `MultiplexedInterleaver` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.MultiplexedInterleaver.getNumInputs

Purpose Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.MultiplexedInterleaver.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `MultiplexedInterleaver` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.MultiplexedInterleaver.reset

Purpose Reset states of the multiplexed interleaver object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `MultiplexedInterleaver` object, `H`.

Purpose

Permute input symbols using a set of shift registers with specified delays

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The input `X` must be a column vector and the data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The multiplexed interleaver object consists of N registers, each with a specified delay. With each new input symbol, a commutator switches to a new register and the new symbol is shifted in while the oldest symbol in that register is shifted out. When the commutator reaches the Nth register, upon the next new input, it returns to the first register.

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.

---

comm.OQPSKDemodulator

Purpose

Demodulate using OQPSK method

Description

The `OQPSKDemodulator` object demodulates a signal that was modulated using the offset quadrature phase shift keying method. The input is a baseband representation of the modulated signal.

Construction

`H = comm.OQPSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the offset quadrature phase shift keying (OQPSK) method.

`H = comm.OQPSKDemodulator(Name,Value)` creates an OQPSK demodulator 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`).

`H = comm.OQPSKDemodulator(PHASE,Name,Value)` creates an OQPSK demodulator object, `H`. This object has the `PhaseOffset` property set to `PHASE` and the other specified properties set to the specified values.

Properties

PhaseOffset

Phase of zeroth point of constellation from $\pi/4$

Specify the phase offset of the zeroth point of the constellation shifted from $\pi/4$, in radians, as a finite, real-valued scalar. The default is `0`.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer values. The default is `false`. When you set this property to `true` the `step` method outputs a column vector of bit values. The vector length must equal to twice the number of demodulated symbols. When you set this property to `false`, the `step` method outputs a column vector. The length of this vector equals to the number of demodulated symbols that contain integer values between `0` and

3. The object produces one output demodulated symbol for each pair of input samples.

OutputDataType

Data type of output

Specify output data type as `Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32`. The default is `Full precision`. When you set this property to `Full precision`, the `step` method output inherits the data type from the input. When the input is of `single` or `double` data, the `step` method outputs the same data type as the input. When the input data is of a fixed-point type, then the `step` method outputs the same data type as if you set the `OutputDataType` property to `Smallest unsigned integer`. When you set the `BitOutput` property to `true`, the `logical` data type becomes a valid option.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in `Smallest unsigned integer` or `Full precision` mode.

Fixed-Point Properties

DerotateFactorDataType

Data type of derotate factor

Specify derotate factor data type as one of `Same word length as input | Custom`. The default is `Same word length as input`. The object uses the derotate factor in the computations only when the step method input is of a fixed-point type and the `PhaseOffset` property has a value that is not a multiple of $\pi/2$.

CustomDerotateFactorDataType

Fixed-point data type of derotate factor

Specify the derotate factor fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is numerictype([],16). This property applies when you set the DerotateFactorDataType property to Custom.

AccumulatorDataType

Data type of accumulator

Specify AccumulatorMode as one of Full precision | Same as input | Custom. The default is Full precision.

CustomAccumulatorDataType

Fixed-point data type of accumulator

Specify the accumulator output fixed-point type as a scaled numerictype object with a signedness of Auto. The default is numerictype([],32,15). This property applies when you set the AccumulatorDataType property to Custom.

AccumulatorRoundingMethod

Rounding of fixed-point numeric value of accumulator

Specify the accumulator rounding method as Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero. The default is Floor.

AccumulatorOverflowAction

Action when fixed-point numeric value of accumulator overflows

Specify the accumulator overflow action as Wrap | Saturate. The default is Wrap.

MappingDataType

Data type of mapping

Specify the mapping data type as Same as accumulator | Custom. The default is Same as accumulator.

CustomMappingDataType

Fixed-point data type of mapping

Specify the mapping fixed-point type as a scaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32,15)`. This property applies when you set the `MappingDataType` property to `Custom`.

Methods

clone	Create OQPSK demodulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset demodulator state
step	Demodulate using OQPSK method

Examples

Modulate and demodulate a signal using OQPSK modulation with a constellation with pi/8 radians of phase offset.

```
   hMod = comm.OQPSKModulator(pi/8);
   hAWGN = comm.AWGNChannel('NoiseMethod', ...
            'Signal to noise ratio (SNR)','SNR',6);
   hDemod = comm.OQPSKDemodulator(pi/8);
% Create an error rate calculator, account for the one symbol delay
   hError = comm.ErrorRate('ReceiveDelay',1);
   for counter = 1:100
   % Transmit a 50-symbol frame
```

```
    data = randi([0 3],50,1);
    modSignal = step(hMod, data);
    noisySignal = step(hAWGN, modSignal);
    receivedData = step(hDemod, noisySignal);
    errorStats = step(hError, data, receivedData);
  end
  fprintf('Error rate = %f\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the OQPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.OQPSKModulator` | `comm.QPSKDemodulator`

Purpose

Create OQPSK demodulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `OQPSKDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.OQPSKDemodulator.constellation

Purpose

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.OQPSKDemodulator

Create a `comm.OQPSKDemodulator` System object, and then calculate its ideal signal constellation.

Create a `comm.OQPSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.OQPSKDemodulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.OQPSKDemodulator

Create a `comm.OQPSKDemodulator` System object, and then plot the ideal signal constellation.

Create a `comm.OQPSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.OQPSKDemodulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.OQPSKDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `OQPSKDemodulator` 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.

comm.OQPSKDemodulator.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 demodulator state

Syntax `reset(H)`

Description `reset(H)` resets the states of the `OQPSKDemodulator` object, `H`.

comm.OQPSKDemodulator.step

Purpose

Demodulate using OQPSK method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` demodulates data, `X`, with the OQPSK demodulator object, `H`, and returns `Y`. Input `X` must be a double, single, or signed fixed-point data type scalar or column vector. The object produces one output symbol for each pair of input samples. When used with the OQPSK modulator object, the step method output has a one symbol delay as compared to the input of the modulator. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

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

Modulate using OQPSK method

Description

The OQPSKModulator object modulates using the offset quadrature phase shift keying method. The output is a baseband representation of the modulated signal.

Construction

H = comm.OQPSKModulator creates a modulator System object, H. This object modulates the input signal using the offset quadrature phase shift keying (OQPSK) method.

H = comm.OQPSKModulator(Name,Value) creates an OQPSK modulator 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).

H = comm.OQPSKModulator(PHASE,Name,Value) creates an OQPSK modulator object, H. This object has the PhaseOffset property set to PHASE and the other specified properties set to the specified values.

Properties

PhaseOffset

Phase of zeroth point of constellation from $\pi/4$

Specify the phase offset of the zeroth point of the constellation shifted from $\pi/4$, in radians, as a finite, real-valued scalar. The default is 0.

BitInput

Assume input is bits

Specify whether the input is bits or integers. The default is false. When you set this property to true, the inputs are bit representations of integers between 0 and 3. The input requires a column vector of bit values with length that is an integer multiple of two. When you set this property to false, the input requires a column vector of integer values between 0 and 3.

OutputDataType

Data type of output

Specify the output data type as `double | single | Custom`. The default is `double`.

Fixed-Point Properties

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `OutputDataType` property to `Custom`.

Methods

clone	Create OQPSK modulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset modulator state
step	Modulate using OQPSK method

Examples

Modulate data using OQPSK, and visualize the modulated data in a scatter plot.

```
% Create binary data for 1000, 2-bit symbols
  data = randi([0 1],2000,1);
% Create an OQPSK modulator System object and accept bits as inputs ar
  hModulator = comm.OQPSKModulator(pi/16,'BitInput',true);
% Modulate and plot the data, ignore the first output symbol
  modData = step(hModulator, data);
  scatterplot(modData(2:end))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the OQPSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.OQPSKDemodulator` | `comm.QPSKModulator`

comm.OQPSKModulator.clone

Purpose Create OQPSK modulator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `OQPSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

Purpose

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.OQPSKModulator

Create a `comm.OQPSKModulator` System object, and then calculate its ideal signal constellation.

Create a `comm.OQPSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.OQPSKModulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.OQPSKModulator

Create a `comm.OQPSKModulator` System object, and then plot the ideal signal constellation.

Create a `comm.OQPSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.OQPSKModulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

comm.OQPSKModulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.OQPSKModulator.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `OQPSKModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.OQPSKModulator.reset

Purpose Reset modulator state

Syntax `reset(H)`

Description `reset(H)` resets the states of the `OQPSKModulator` object, `H`.

Purpose

Modulate using OQPSK method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` modulates input data, `X`, with the OQPSK modulator object, `H`, and returns baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with numeric, logical, or fixed-point data types.

The OQPSK modulator object upsamples by a factor of two. The `step` method outputs the length, `Y`, as $2 \times N$, where N is the length of the input, `X`. The `step` method outputs an initial condition of zero, which is unrelated to the input values.

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

Combine inputs using orthogonal space-time block code

Description

The OSTBCCombiner object combines the input signal (from all of the receive antennas) and the channel estimate signal to extract the soft information of the symbols encoded by an OSTBC. The input channel estimate does not need to be constant and can vary at each call to the step method. The combining algorithm uses only the estimate for the first symbol period per codeword block. A symbol demodulator or decoder would follow the Combiner object in a MIMO communications system.

Construction

H = comm.OSTBCCombiner creates an orthogonal space-time block code (OSTBC) combiner System object, H. This object combines the input signal (from all of the receive antennas) with the channel estimate signal to extract the soft information of the symbols encoded by an OSTBC.

H = comm.OSTBCCombiner(Name,Value) creates an OSTBC Combiner 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).

H = comm.OSTBCCombiner(N,M,Name,Value) creates an OSTBC Combiner object, H. This object has the NumTransmitAntennas property set to N, the NumReceiveAntennas property set to N, and the other specified properties set to the specified values.

Properties

NumTransmitAntennas

Number of transmit antennas

Specify the number of antennas at the transmitter as 2 | 3 | 4. The default is 2.

SymbolRate

Symbol rate of code

Specify the symbol rate of the code as 3/4 | 1/2. The default is 3/4. This property applies when the NumTransmitAntennas property is greater than 2. For 2 transmit antennas, the symbol rate defaults to 1.

NumReceiveAntennas

Number of receive antennas

Specify the number of antennas at the receiver as a double-precision, real, scalar integer value from 1 to 8. The default is 1.

Fixed-Point Properties

RoundingMethod

Rounding of fixed-point numeric values

Specify the rounding method as `Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero`. The default is `Floor`.

OverflowAction

Action when fixed-point numeric values overflow

Specify the overflow action as one of `Wrap | Saturate`. The default is `Wrap`. This property specifies the action to be taken in case of overflow. Such overflow occurs if the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result.

ProductDataType

Data type of product

Specify the product data type as one of `Full precision | Custom`. The default is `Full precision`.

CustomProductDataType

Fixed-point data type of product

Specify the product fixed-point type as a scaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32,16)`.

This property applies when you set the ProductDataType property to Custom.

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as Full precision | Same as product | Custom. The default is Full precision.

CustomAccumulatorDataType

Fixed-point data type of accumulator

Specify the accumulator fixed-point type as a scaled numerictype object with a signedness of Auto. The default is numerictype([],32,16). This property applies when you set the AccumulatorDataType property to Custom.

EnergyProductDataType

Data type of energy product

Specify the complex energy product data type as one of Full precision | Same as product | Custom. The default is Full precision. This property sets the data type of the complex product in the denominator to calculate the total energy in the MIMO channel.

CustomEnergyProductDataType

Fixed-point data type of energy product

Specify the energy product fixed-point type as a scaled numerictype object with a signedness of Auto. The default is numerictype([],32,16). This property applies when you set the EnergyProductDataType property to Custom.

EnergyAccumulatorDataType

Data type of energy accumulator

Specify the energy accumulator data type as one of Full precision | Same as energy product | Same as accumulator

| Custom. The default is Full precision. This property sets the data type of the summation in the denominator to calculate the total energy in the MIMO channel.

CustomEnergyAccumulatorDataType

Fixed-point data type of energy accumulator

Specify the energy accumulator fixed-point type as a scaled numerictype object with a signedness of Auto. The default is numerictype([],32,16). This property applies when you set the EnergyAccumulatorDataType property to Custom.

DivisionDataType

Data type of division

Specify the division data type as one of Same as accumulator | Custom. The default is Same as accumulator. This property sets the data type at the output of the division operation. The setting normalizes diversity combining by the total energy in the MIMO channel.

CustomDivisionDataType

Fixed-point data type of division

Specify the division fixed-point type as a scaled numerictype object with a signedness of Auto. The default is numerictype([],32,16). This property applies when you set the DivisionDataType property to Custom.

Methods

clone	Create OSTBC combiner object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method

isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Combine inputs using orthogonal space-time block code

Examples

Encode and decode QPSK modulated data with OSTBC and calculate error.

```
% Define system parameters
    numTx = 2; numRx = 1; Rs = 1e6; maxDopp = 30; ...
    numBits = 1024; SNR = 10;
% Create modulator and encoder System objects
    hMod = comm.QPSKModulator(...
            'BitInput',      true,...
            'SymbolMapping', 'Gray');
    hDemod = comm.QPSKDemodulator(...
            'SymbolMapping', 'Gray',...
            'BitOutput',      true);
    hOSTBCEnc = comm.OSTBCEncoder(...
            'NumTransmitAntennas', numTx);
    hOSTBCComb = comm.OSTBCCombiner(...
            'NumTransmitAntennas', numTx,...
            'NumReceiveAntennas',  numRx);
% Create MIMO channel System object
    hChan = comm.MIMOChannel(...
            'SampleRate',                Rs,...
            'MaximumDopplerShift',       maxDopp,...
            'NumTransmitAntennas',       numTx,...
            'NumReceiveAntennas',        numRx,...
            'ReceiveCorrelationMatrix', 1,...
            'PathGainsOutputPort',       true);
% Create AWGN channel System object
    hAWGN = comm.AWGNChannel(...
```

```
            'NoiseMethod', 'Signal to noise ratio (SNR)',...
            'SNR',         SNR,...
            'SignalPower', 1);

% Generate data
    data = randi([0 1], numBits, 1);
% Modulate data
    modData = step(hMod, data);
% Encode modulated data using OSTBC
    encData = step(hOSTBCEnc, modData);
% Transmit through Rayleigh and AWGN channels
    [chanOut, pathGains] = step(hChan, encData);
    rxSignal = step(hAWGN, chanOut);
% Decode and demodulate received signal
    decData = step(hOSTBCComb, rxSignal, squeeze(pathGains));
    receivedData = step(hDemod, decData);
% Compute number of bit errors in received data
    errors = biterr(data, receivedData);
    fprintf(1, ['\nThere were %d errors in the received signal ' ...
          'out of %d bits transmitted\n'], errors, length(data));
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the OSTBC Combiner block reference page. The object properties correspond to the block parameters.

See Also

`comm.OSTBCEncoder`

comm.OSTBCCombiner.clone

Purpose Create OSTBC combiner object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `OSTBCCombiner` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.OSTBCCombiner.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `OSTBCCombiner` 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.

comm.OSTBCCombiner.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Combine inputs using orthogonal space-time block code

Syntax

```
Y = step(H,X,CEST)
```

Description

`Y = step(H,X,CEST)` combines the received data, `X`, and the channel estimate, `CEST`, to extract the symbols encoded by an OSTBC. Both `X` and `CEST` are complex-valued and of the same data type, which can be double, single, or signed fixed point with power-of-two slope and zero bias. When the step method input `X` has double or single precision, the output, `Y`, has the same data type as the input. The input channel estimate can remain constant or can vary during each codeword block transmission. The combining algorithm uses the estimate only for the first symbol period per codeword block.

The time domain length, T/`SymbolRate`, must be a multiple of the codeword block length. T is the output symbol sequence length in the time domain. Specifically, when you set the `NumTransmitAntennas` property to `2`, T/`SymbolRate` must be a multiple of two. When you set the `NumTransmitAntennas` property greater than `2`, T/`SymbolRate` must be a multiple of four. For an input of T/`SymbolRate` rows by `NumReceiveAntennas` columns, the input channel estimate, `CEST`, must be a matrix of size T/`SymbolRate`by NumTransmitAntennas by NumReceiveAntennas. In this case, the extracted symbol data, `Y`, is a column vector with T elements. Input matrix size can be F by T/`SymbolRate` by `NumReceiveAntennas`, where F is an optional dimension (typically frequency domain) over which the combining calculation is independent. In this case, the input channel estimate, `CEST`, must be a matrix of size F by T/`SymbolRate` by `NumTransmitAntennas` by `NumReceiveAntennas`. The extracted symbol data, `Y`, is an F rows by T columns 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

Encode input using orthogonal space-time block code

Description

The `OSTBCEncoder` object encodes an input symbol sequence using orthogonal space-time block code (OSTBC). The block maps the input symbols block-wise and concatenates the output codeword matrices in the time domain.

Construction

`H = comm.OSTBCEncoder` creates an orthogonal space-time block code (OSTBC) encoder System object, `H`. This object maps the input symbols block-wise and concatenates the output codeword matrices in the time domain.

`H = comm.OSTBCEncoder(Name,Value)` creates an OSTBC encoder 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`).

`H = comm.OSTBCEncoder(N,Name,Value)` creates an OSTBC encoder object, `H`. This object has the `NumTransmitAntennas` property set to `N`, and the other specified properties set to the specified values.

Properties

NumTransmitAntennas

Number of transmit antennas

Specify the number of antennas at the transmitter as `2 | 3 | 4`. The default is `2`.

SymbolRate

Symbol rate of code

Specify the symbol rate of the code as one of `3/4 | 1/2`. The default is `3/4`. This property applies when you set the `NumTransmitAntennas` property to greater than `2`. For `2` transmit antennas, the symbol rate defaults to `1`.

Fixed-Point Properties

OverflowAction

Action when fixed-point numeric values overflow

Specify the overflow action as one of `Wrap | Saturate`. The default is `Wrap`. This property specifies the action to be taken in the case of an overflow. Such overflow occurs when the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result.

Methods

clone	Create OSTBC encoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Encode input using orthogonal space-time block code

Examples

Encode BPSK modulated data with OSTBC.

```
% Generate random binary data
   data = randi([0 1], 8, 1);

%Create BPSK Modulator and obtain modulated data
   hMod = comm.BPSKModulator;
   modData = step(hMod, data);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the OSTBC Encoder block reference page. The object properties correspond to the block parameters.

When this object processes variable-size signals:

- If the input signal is a column vector, the first dimension can change, but the second dimension must remain fixed at `1`.
- If the input signal is a matrix, both dimensions can change.

See Also

`comm.OSTBCCombiner`

Purpose Create OSTBC encoder object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `OSTBCEncoder` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.OSTBCEncoder.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `OSTBCEncoder` 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.

comm.OSTBCEncoder.release

Purpose

Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 Encode input using orthogonal space-time block code

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` encodes the input data, `X`, using OSTBC encoder object, `H`. The input is a complex-valued column vector or matrix of data type double, single, or signed fixed-point with power-of-two slope and zero bias. The step method output, `Y`, is the same data type as the input data. The time domain length, T, of `X` must be a multiple of the number of symbols in each codeword matrix. Specifically, when you set the `NumTransmitAntennas` property is `2` or the `SymbolRate` property is `1/2`, T must be a multiple of two and when the `SymbolRate` property to `3/4`, T must be a multiple of three. For a time or spatial domain input of T rows by one column, the encoded output data, `Y`, is a (T/`SymbolRate`)-by-`NumTransmitAntennas` matrix. The input matrix size can be F rows by T columns, where F is the additional dimension (typically the frequency domain) over which the encoding calculation is independent. In this case, the output is an F-by-(T/`SymbolRate`)-by-`NumTransmitAntennas` 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.

---

comm.OVSFCode

Purpose

Generate OVSF code

Description

The `OVSFCode` object generates an OVSF code from a set of orthogonal codes. OVSF codes were first introduced for 3G communication systems. They are primarily used to preserve orthogonality between different channels in a communication system.

Construction

`H = comm.OVSFCode` creates an orthogonal variable spreading factor (OVSF) code generator System object, `H`. This object generates an OVSF code.

`H = comm.OVSFCode(Name,Value)` creates an OVSF code generator 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

SpreadingFactor

Length of generated code

Specify the length of the generated code as an integer scalar value with a power of two. The default is `64`.

Index

Index of code of interest

Specify the index of the desired code from the available set of codes that have the spreading factor specified in the `SpreadingFactor` property. This property must be an integer scalar in the range `0` to `SpreadingFactor–1`. The default is `60`.

OVSF codes are defined as the rows of an *n*-by-*n* matrix, *Cn*, where *n* is the value specified in the `SpreadingFactor` property.

You can define the matrix *Cn* recursively as follows:

First, define $C1$ = `[1]`.
Next, assume that *Cn* is defined and let *Cn*(*k*) denote the *k*-th row of *Cn*.

Then, $C2n$ = [Cn(0) Cn(0); Cn(0) -Cn(0); ... ; Cn(n-1) Cn(n–1); Cn(n–1)–Cn(n–1)].

Cn is only defined for values of n that are a power of 2. Set the this property to a value of k to choose the k-th row of the C matrix as the code of interest.

SamplesPerFrame

Number of output samples per frame

Specify the number of OVSF code samples that the `step` method outputs as a numeric, positive, integer scalar value. The default is `1`. If you set this property to a value of M, then the `step` method outputs M samples of an OVSF code of length N. N is the length of the OVSF code that you specify in the `SpreadingFactor` property.

OutputDataType

Data type of output

Specify output data type as one of `double | int8`. The default is `double`.

Methods

clone	Create OVSF code generator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of OVSF code generator object
step	Generate OVSF code

Examples

Generate 10 samples of an OVSF code with a spreading factor of 64.

```
hOVSF = comm.OVSFCode('SamplesPerFrame', 10,'SpreadingFactor',64);
seq = step(hOVSF)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the OVSF Code Generator block reference page. The object properties correspond to the block parameters, except:

- The object does not have a property to select frame based outputs.
- The object does not have a property that corresponds to the **Sample time** parameter.

See Also

`comm.HadamardCode` | `comm.WalshCode`

Purpose

Create OVSF code generator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `OVSFCode` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.OVSFCode.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.OVSFCode.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `OVSFCode` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.OVSFCode.reset

Purpose Reset states of OVSF code generator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `OVSFCode` object, `H`.

Purpose

Generate OVSF code

Syntax

```
Y = step(H)
```

Description

`Y = step(H)` outputs a frame of the OVSF code in column vector `Y`. Specify the frame length with the `SamplesPerFrame` 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.

---

comm.PAMDemodulator

Purpose

Demodulate using M-ary PAM method

Description

The `PAMDemodulator` object demodulates a signal that was modulated using M-ary pulse amplitude modulation. The input is a baseband representation of the modulated signal.

Construction

`H = comm.PAMDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the M-ary pulse amplitude modulation (M-PAM) method.

`H = comm.PAMDemodulator(Name,Value)` creates an M-PAM demodulator 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`).

`H = comm.PAMDemodulator(M,Name,Value)` creates an M-PAM demodulator object, `H`. This object has the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is `4`. When you set the `BitOutput` property to `false`, this value must be even. When you set the `BitOutput` property to `true`, this value requires an integer power of two.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. The default is `false`.

When you set this property to `true` the `step` method outputs a column vector of bit values with length equal to **log2**(`ModulationOrder`) times the number of demodulated symbols.

When you set this property to `false`, the `step` method outputs a column vector, with length equal to the input data vector. This value contains integer symbol values between `0` and `ModulationOrder`–1.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder`) bits to the corresponding symbol as one of `Binary | Gray`. The default is `Gray`.

When you set this property to `Gray`, the object uses a Gray-encoded signal constellation.

When you set this property to `Binary`, the integer m, between $0 \leq m \leq$(`ModulationOrder`–1) maps to the complex value $2m$-`ModulationOrder+1`.

NormalizationMethod

Constellation normalization method

Specify the method used to normalize the signal constellation as one of `Minimum distance between symbols | Average power | Peak power`. The default is `Minimum distance between symbols`.

MinimumDistance

Minimum distance between symbols

Specify the distance between two nearest constellation points as a positive, real, numeric scalar value. The default is `2`. This property applies when you set the `NormalizationMethod` property to `Minimum distance between symbols`.

AveragePower

Average power of constellation

Specify the average power of the symbols in the constellation as a positive, real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` property to `Average power`.

PeakPower

Peak power of constellation

Specify the maximum power of the symbols in the constellation as a positive, real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` property to `Peak power`.

OutputDataType

Data type of output

Specify the output data type as one of `Full precision` | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`. The default is `Full precision`.

When you set this property to `Full precision`, and the input data type is single or double precision, the output data has the same data type that of the input.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in `Smallest unsigned integer` or `Full precision` mode.

When the input data is of a fixed-point type, the output data type behaves as if you had set the `OutputDataType` property to `Smallest unsigned integer`.

When you set the `BitOutput` property to `true`, then `logical` data type becomes a valid option.

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".

DenormalizationFactorDataType

Data type of denormalization factor

Specify the denormalization factor data type as one of `Same word length as input | Custom`. The default is `Same word length as input`.

CustomDenormalizationFactorDataType

Fixed-point data type of denormalization factor

Specify the denormalization factor fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `DenormalizationFactorDataType` property to `Custom`.

ProductDataType

Data type of product

Specify the product data type as one of `Full precision | Custom`. The default is `Full precision`. When you set this property to `Full precision` the object calculates the full-precision product word and fraction lengths. This property applies when you set the `FullPrecisionOverride` property to false.

CustomProductDataType

Fixed-point data type of product

Specify the product fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32)`. This property applies when you set the `FullPrecisionOverride` property to false and the `ProductDataType` property to `Custom`.

ProductRoundingMethod

Rounding of fixed-point numeric value of product

Specify the product rounding method as one of `Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero`. The default is `Floor`. This property applies when the object is not in a full precision configuration

ProductOverflowAction

Action when fixed-point numeric value of product overflows

Specify the product overflow action as one of `Wrap | Saturate`. The default is `Wrap`. This property applies when the object is not in a full precision configuration.

SumDataType

Data type of sum

Specify the sum data type as one of `Full precision | Same as product | Custom`. The default is `Full precision`. When you set this property to `Full precision`, the object calculates the full-precision sum word and fraction lengths. This property applies when you set the `FullPrecisionOverride` property to false

CustomSumDataType

Fixed-point data type of sum

Specify the sum fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32)`. This property applies when you set the the `FullPrecisionOverride` property to false and the `SumDataType` property to `Custom`.

Methods

clone	Create M-PAM demodulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Demodulate using M-ary PAM method

Examples

Modulate and demodulate a signal using 16-PAM modulation.

```
hMod = comm.PAMModulator(16);
hAWGN = comm.AWGNChannel('NoiseMethod', ...
                'Signal to noise ratio (SNR)', ...
                'SNR',20, 'SignalPower', 85);
hDemod = comm.PAMDemodulator(16);
%Create an error rate calculator
hError = comm.ErrorRate;
for counter = 1:100
  % Transmit a 50-symbol frame
  data = randi([0 hMod.ModulationOrder-1],50,1);
  modSignal = step(hMod, data);
  noisySignal = step(hAWGN, modSignal);
  receivedData = step(hDemod, noisySignal);
  errorStats = step(hError, data, receivedData);
end
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
```

```
errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PAM Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.PAMModulator`

Purpose

Create M-PAM demodulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `PAMDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.PAMDemodulator.constellation

Purpose

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.PAMDemodulator

Create a `comm.PAMDemodulator` System object, and then calculate its ideal signal constellation.

Create a `comm.PAMDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.PAMDemodulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.PAMDemodulator

Create a `comm.PAMDemodulator` System object, and then plot the ideal signal constellation.

Create a `comm.PAMDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.PAMDemodulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.PAMDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `PAMDemodulator` 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.

comm.PAMDemodulator.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Demodulate using M-ary PAM method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` demodulates data, `X`, with the M-PAM demodulator System object, `H`, and returns `Y`. Input `X` must be a scalar or column vector. The data type of the input can be double or single precision, signed integer, or signed fixed point (fi objects). Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

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.

---

comm.PAMModulator

Purpose

Modulate using M-ary PAM method

Description

The `PAMModulator` object modulates using M-ary pulse amplitude modulation. The output is a baseband representation of the modulated signal. The M-ary number parameter, M, represents the number of points in the signal constellation and requires an even integer.

Construction

`H = comm.PAMModulator` creates a modulator System object, `H`. This object modulates the input signal using the M-ary pulse amplitude modulation (M-PAM) method.

`H = comm.PAMModulator(Name,Value)` creates an M-PAM modulator 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`).

`H = comm.PAMModulator(M,Name,Value)` creates an M-PAM modulator object, `H`. This object has the `ModulationOrder` property set to `M` and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is `4`. When you set the `BitInput` property to `false`, ModulationOrder must be even. When you set the `BitInput` property to `true`, ModulationOrder must be an integer power of two.

BitInput

Assume bit inputs

Specify whether the input is in bits or integers. The default is `false`.

When you set this property to `true`, the `step` method input requires a column vector of bit values whose length is an integer multiple of **log2**(ModulationOrder). This vector contains bit representations of integers between 0 and ModulationOrder–1.

When you set this property to false, the step method input must be a column vector of integer symbol values between 0 and ModulationOrder–1.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(ModulationOrder) input bits to the corresponding symbol as one of Binary | Gray. The default is Gray.

When you set this property to Gray, the object uses a Gray-encoded signal constellation.

When you set this property to Binary, the input integer m, between $0 \leq m \leq$ ModulationOrder-1) maps to the complex value $2m$– ModulationOrder + 1.

NormalizationMethod

Constellation normalization method

Specify the method used to normalize the signal constellation as one of Minimum distance between symbols | Average power | Peak power. The default is Minimum distance between symbols.

MinimumDistance

Minimum distance between symbols

Specify the distance between two nearest constellation points as a positive, real, numeric scalar value. The default is 2. This property applies when you set the NormalizationMethod property to Minimum distance between symbols.

AveragePower

Average power of constellation

Specify the average power of the symbols in the constellation as a positive, real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` property to `Average power`.

PeakPower

Peak power of constellation

Specify the maximum power of the symbols in the constellation as a positive, real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` property to `Peak power`.

OutputDataType

Data type of output

Specify the output data type as one of `double | single | Custom`. The default is `double`.

Fixed-Point Properties

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `OutputDataType` property to `Custom`.

Methods

clone	Create PAM modulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Modulate using M-ary PAM method

Examples

Modulate data using 16-PAM modulation, and visualize the data in a scatter plot.

```
% Create binary data for 100, 4-bit symbols
data = randi([0 1],400,1);
% Create a 16-PAM modulator System object with bits as inputs and
% Gray-coded signal constellation
hModulator = comm.PAMModulator(16,'BitInput',true);
% Modulate and plot the data
modData = step(hModulator, data);
constellation(hModulator)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PAM Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.PAMDemodulator`

Purpose

Create PAM modulator object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `PAMModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

Purpose

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.PAMModulator

Create a `comm.PAMModulator` System object, and then calculate its ideal signal constellation.

Create a `comm.PAMModulator` System object by entering the following at the MATLAB command line:

```
h = comm.PAMModulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.PAMModulator

Create a `comm.PAMModulator` System object, and then plot the ideal signal constellation.

Create a `comm.PAMModulator` System object by entering the following at the MATLAB command line:

```
h = comm.PAMModulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.PAMModulator.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `PAMModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Modulate using M-ary PAM method

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` modulates input data, `X`, with the M-PAM modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with numeric, logical, or fixed-point data types.

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

Apply phase and frequency offsets to input signal

Description

The `PhaseFrequencyOffset` object applies phase and frequency offsets to an incoming signal.

Construction

`H = comm.PhaseFrequencyOffset` creates a phase and frequency offset System object, `H`. This object applies phase and frequency offsets to an input signal.

`H = comm.PhaseFrequencyOffset(Name,Value)` creates a phase and frequency offset 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

PhaseOffset

Phase offset

Specify the phase offset in degrees. The default is `0`. If the `step` method input is an *M*-by-*N* matrix, the `PhaseOffset` property can be set to a numeric scalar, an *M*-by-1, or 1-by-*N* numeric vector, or an *M*-by-*N* numeric matrix.

When you set the `PhaseOffset` property to a scalar value, the object applies the constant specified phase offset to each column of the input matrix.

When you set this property to an *M*-by-1 vector, the object applies time varying phase offsets, specified in the vector of this property, to each column of the input to the `step` method.

When you set this property to a 1-by-*N* vector, the object applies the *i*-th constant phase offset of this property to the *i*-th column of the input to the `step` method.

When you set this property to an *M*-by-*N* matrix, the object applies the *i*-th time varying phase offsets, specified in the *i*-th column of this property, to the *i*-th column of the input to the `step` method. This property is tunable.

FrequencyOffsetSource

Source of frequency offset

Specify the source of the frequency offset as one of `Property | Input port`. The default is `Property`. If you set this property to `Property`, you can specify the frequency offset using the `FrequencyOffset` property. If you set this property to `Input port`, you specify the frequency offset as a step method input.

FrequencyOffset

Frequency offset

Specify the frequency offset in Hertz. The default is `0`. If the `step` method input is an M-by-N matrix, then the `FrequencyOffset` property is a numeric scalar, an M-by-1, or 1-by-N numeric vector, or an M-by-N numeric matrix.

This property applies when you set the `FrequencyOffsetSource` property to `Property`.

When you set this property to a scalar value, the object applies the constant specified frequency offset to each column of the input to the `step` method.

When you set this property to an M-by-1 vector, the object applies time-varying frequency offsets. These offsets are specified in the property, to each column of the input to the `step` method.

When you set this property to a 1-by-N vector, the object applies the i-th constant frequency offset in this property to the i-th column of the input to the `step` method.

When you set this property to an M-by-N matrix, the object applies the i-th time varying frequency offset. This offset is specified in the i-th column of this property and to the i-th column of input to the `step` method. This property is tunable.

SampleRate

Sample rate

Specify the sample rate of the input samples in seconds as a double-precision, real, positive scalar value. The default is `1`.

`SampleRate` = Input Vector Size / Simulink Sample Time

Methods

clone	Create phase and frequency offset object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Apply phase and frequency offsets to input signal

Examples

Introduce phase offset to a rectangular 16-QAM signal.

```
    data = (0:15)';
    M = 16; % Modulation order
    hMod = comm.RectangularQAMModulator(M);
    hPFO = comm.PhaseFrequencyOffset('PhaseOffset', 20,'SampleRate',1e
% Modulate data
    modData = step(hMod, data);
    scatterplot(modData);
    title(' Original Constellation');xlim([-5 5]);ylim([-5 5])
% Introduce phase offset
    impairedData = step(hPFO,modData);
    scatterplot(impairedData);
    title('Constellation after phase offset');xlim([-5 5]);ylim([-5 5]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Phase/Frequency Offset block reference page. The object properties correspond to the block parameters, except:

> The object provides a `SampleRate` property, which you must specify. The block senses the sample time of the signal and therefore does not have a corresponding parameter.

See Also

`comm.ThermalNoise` | `comm.PhaseNoise` | `comm.MemorylessNonlinearity`

Purpose

Create phase and frequency offset object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `PhaseFrequencyOffset` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.PhaseFrequencyOffset.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.PhaseFrequencyOffset.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `PhaseFrequencyOffset` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.PhaseFrequencyOffset.step

Purpose

Apply phase and frequency offsets to input signal

Syntax

```
Y = step(H,X)
Y = step(H,X,FRQ)
```

Description

`Y = step(H,X)` applies phase and frequency offsets to input `X`, and returns `Y`. The input `X` is a double or single precision matrix `X`, of dimensions MxN. M is the number of time samples in the input signals and N is number of channels. Both M and N can be equal to 1. The object adds phase and frequency offsets independently to each column of `X`. The data type and dimensions of `X` and `Y` are the same.

`Y = step(H,X,FRQ)` uses `FRQ` as the frequency offset that the object applies to input `X` when you set the `FrequencyOffsetSource` property to 'Input port'. When the `X` input is an MxN matrix, the value for `FRQ` can be a numeric scalar, an Mx1 or 1xN numeric vector, or an MxN numeric matrix. When the `FRQ` input is a scalar, the object applies a constant frequency offset, `FRQ`, to each column of `X`. When the `FRQ` input is an Mx1 vector, the object applies time varying frequency offsets, which are specified in the `FRQ` vector, to each column of `X`. When the `FRQ` input is a 1xN vector, the object applies the ith constant frequency offset in `FRQ` to the ith column of `X`. When the `FRQ` input is an MxN matrix, the object applies the ith time varying frequency offsets, specified in the ith column of `FRQ`, to the ith column of `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

Apply phase noise to complex, baseband signal

Description

The `PhaseNoise` object applies phase noise to a complex, baseband signal. This phase noise has a 1/*f* spectral characteristic over its entire frequency range.

Construction

`H = comm.PhaseNoise` creates a phase noise System object, `H`. This object applies phase noise with the specified level at the specified frequency offset to a complex, baseband input signal.

`H = comm.PhaseNoise(Name,Value)` creates a phase noise 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`).

`H = comm.PhaseNoise(LEVEL,OFFSET,Name,Value)` creates a phase noise object, `H`. This object has the `Level` property set to `LEVEL`, the `FrequencyOffset` property set to `OFFSET` and the other specified properties set to the specified values.

Properties

Level

Phase noise level

Specify the phase noise level in decibels relative to carrier per Hertz (dBc/Hz) at a frequency offset specified by the `FrequencyOffset` property. The default is `[-60 -80]`. This property requires a negative, real scalar or vector of data type `double`.

FrequencyOffset

Frequency offset

Specify the frequency offset in Hertz as a nonnegative, real scalar or increasing vector of data type `double`. The default is `[20 200]`.

SampleRate

Sample rate

Specify the sample rate in Hertz as a positive, real scalar or vector of data type double. The default is `1024`. The System object does not use this property when you specify `Level` and `FrequencyOffset` as scalars.

Methods

clone	Create phase noise object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Apply phase noise to a complex, baseband signal

Examples

Add a phase noise vector and frequency offset vector to a 16-QAM signal. Then, plot the signal.

```
% Create 16-QAM modulator
    hMod = comm.RectangularQAMModulator(16, ...
        'NormalizationMethod','Average power', 'AveragePower',10);
% Create phase noise System object
    hPhNoise = comm.PhaseNoise('Level',[-60 -80], ...
        'FrequencyOffset',[20 200], ...
        'SampleRate',1024);
% Generate modulated symbols
    modData = step(hMod, randi([0 15], 1000, 1));
% Apply phase noise and plot the result
    y = step(hPhNoise, modData);
    scatterplot(y)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Phase Noise block reference page. The object properties correspond to the block parameters, except:

- The object respects the data types and does perform any casting other than casting the output to the input data type. The result of **exp**($1i \times$ *phase_noise*) is cast to the input data type first, before multiplying with the input signal. This order prevents the output (phase distorted) signal from being downcast to single precision if any of the properties are of data type single while the input data type is double precision.
- This object uses the MATLAB default random stream to generate random numbers. The block uses a random number generator based on the V5 RANDN (Ziggurat) algorithm. In addition, the block uses an initial seed, set with the **Initial seed** parameter to initialize the random number generator. Every time the system that contains the block runs, the block generates the same sequence of random numbers. To generate reproducible numbers using this object, reset the MATLAB default random stream using the following code.

  ```
  reset(RandStream.getGlobalStream)
  ```

  For more information, see help for `RandStream`.

See Also

`comm.PhaseFrequencyOffset` | `comm.MemorylessNonlinearity`

Purpose

Create phase noise object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `PhaseNoise` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.PhaseNoise.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `PhaseNoise` 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.

comm.PhaseNoise.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Apply phase noise to a complex, baseband signal

Syntax

`Y = step(H,X)`

Description

`Y = step(H,X)` adds phase noise with the specified level, at the specified frequency offset, to the input `X` and returns the result in `Y`. `X` must be a complex scalar or column vector of data type double or single. The `step` method outputs, `Y`, with the same data type and dimensions 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.

---

comm.PNSequence

Purpose

Generate a pseudo-noise (PN) sequence

Description

The PNSequence object generates a sequence of pseudorandom binary numbers using a linear-feedback shift register (LFSR). This block implements LFSR using a simple shift register generator (SSRG, or Fibonacci) configuration. You can use a pseudonoise sequence in a pseudorandom scrambler and descrambler. You can also use one in a direct-sequence spread-spectrum system.

Construction

`H = comm.PNSequence` creates a pseudo-noise (PN) sequence generator System object, `H`. This object generates a sequence of pseudorandom binary numbers using a linear-feedback shift register (LFSR).

`H = comm.PNSequence(Name,Value)` creates a PN sequence generator 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

Polynomial

Generator polynomial

Specify the polynomial that determines the shift register's feedback connections. The default is `[1 0 0 0 0 1 1]`. You can specify the generator polynomial as a numeric, binary vector that lists the coefficients of the polynomial in descending order of powers. The first and last elements must equal `1`, and the length of this vector must be n+1. The value n indicates the degree of the generator polynomial. Alternatively, you can specify the generator polynomial as a numeric vector containing the exponents of z for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`. For example, `[1 0 0 0 0 0 1 0 1]` and `[8 2 0]` represent the same polynomial, $g(z) = z^8 + z^2 + 1$.

The PN sequence has a period of $N = 2^n - 1$.

InitialConditionsSource

Source of initial conditions

Specify the source of the initial conditions that determines the start of the PN sequence as one of `Property | Input port`. The default is `Property`. When you set this property to `Property`, the initial conditions can be specified as a scalar or binary vector using the `InitialConditions` property. When you set this property to `Input port`, you specify the initial conditions as an input to the `step` method. The object accepts a binary scalar or a binary vector input. The length of the input must equal the degree of the generator polynomial that the `Polynomial` property specifies.

InitialConditions

Initial conditions of shift register

Specify the initial values of the shift register as a binary, numeric scalar or a binary, numeric vector. The default is `[0 0 0 0 0 1]`. Set the vector length equal to the degree of the generator polynomial. If you set this property to a vector, each element of the vector corresponds to the initial value of the corresponding cell in the shift register. If you set this property to a scalar, the initial conditions of all the cells of the shift register are the specified scalar value. The scalar, or at least one element of the specified vector, must be nonzero for the object to generate a nonzero sequence.

MaskSource

Source of mask to shift PN sequence

Specify the source of the mask that determines the shift of the PN sequence as one of `Property | Input port`. The default is `Property`. When you set this property to `Property`, the mask can be specified as a scalar or binary vector using the `Mask` property. When you set this property to `Input port`, the mask, which is an input to the `step` method, can only be specified as a binary vector. This vector must have a length equal to the degree of the generator polynomial specified in the `Polynomial` property.

Mask

Mask to shift PN sequence

Specify the mask that determines how the PN sequence is shifted from its starting point as a numeric, integer scalar or as a binary vector. The default is `0`.

When you set this property to an integer scalar, the value is the length of the shift. A scalar shift can be positive or negative.

When the PN sequence has a period of $N = 2^n - 1$, where n is the degree of the generator polynomial that you specify in the `Polynomial` property, the object wraps shift values that are negative or greater than N.

When you set this property to a binary vector, its length must equal the degree of the generator polynomial specified in the `Polynomial` property. The mask vector that represents $m(z) = z^D$ modulo *g(z)*, where *g(z)* is the generator polynomial, and the mask vector corresponds to a shift of D. For example, for a generator polynomial of degree of `4`, the mask vector corresponding to D = `2` is `[0 1 0 0]`, which represents the polynomial $m(z) = z^2$.

You can calculate the mask vector using the `shift2mask` function. This property applies when you set the `MaskSource` property to `Property`.

VariableSizeOutput

Enable variable-size outputs

Set this property to true to enable an additional input to the step method. The default is false. When you set this property to true, the enabled input specifies the output size of the PN sequence used for the step. The input value must be less than or equal to the value of the `MaximumOutputSize` property.

When you set this property to false, the `SamplesPerFrame` property specifies the number of output samples.

MaximumOutputSize

Maximum output size

Specify the maximum output size of the PN sequence as a positive integer 2-element row vector. The second element of the vector must be `1`. The default is [10 1].

This property applies when you set the `VariableSizeOutput` property to true.

SamplesPerFrame

Number of outputs per frame

Specify the number of PN sequence samples that the step method outputs as a numeric, positive, integer scalar value. The default is `1`. If you set this property to a value of M, then the `step` method outputs M samples of a PN sequence that has a period of $N = 2^n - 1$. The value n represents the degree of the generator polynomial that you specify in the `Polynomial` property. If you set the `BitPackedOutput` property to `false`, the samples are bits from the PN sequence. If you set the `BitPackedOutput` property to `true`, then the output corresponds to `SamplesPerFrame` groups of bit-packed samples.

ResetInputPort

Enable generator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. This input resets the states of the PN sequence generator to the initial conditions specified in the `InitialConditions` property.

BitPackedOutput

Output integer representations of bit-packed words

Set this property to `true` to enable bit-packed outputs. The default is `false`. In this case, the `step` method outputs a column vector of length M, which contains integer representations of bit words of length P. M is the number of samples per frame specified in the `SamplesPerFrame` property. P is the size of the bit-packed words specified in the NumPackedBits property. The first bit from the left in the bit-packed word is considered the most significant bit.

NumPackedBits

Number of bits per bit-packed word

Specify the number of bits to pack into each output data word as a numeric, integer scalar value between 1 and 32. The default is 8. This property applies when you set the BitPackedOutput property to true.

SignedOutput

Output signed bit-packed words

Set this property to true to obtain signed, bit-packed, output words. The default is false. In this case, a 1 in the most significant bit (sign bit) indicates a negative value. The property indicates negative numbers in a two's complement format. This property applies when you set the BitPackedOutput property to true.

OutputDataType

Data type of output

Specify the output data type as one of double | logical | Smallest unsigned integer when the BitPackedOutput property is false. The default is double. Specify the output data type as double | Smallest unsigned integer when the BitPackedOutput property is set to true.

You must have a Fixed-Point Designer user license to use this property in Smallest unsigned integer mode.

Methods

clone	Create PN sequence generator object with same property values
getNumInputs	Number of expected inputs to step method

getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of PN sequence generator object
step	Generate a pseudo-noise (PN) sequence

Examples

Get 2 frames of 14 samples of a maximal length PN-sequence of period 2^3–1 (i.e., get two periods of the sequence).

```
hpn = comm.PNSequence('Polynomial',[3 2 0], ...
        'SamplesPerFrame', 14, 'InitialConditions',[0 0 1]);
x1 = step(hpn);
x2 = step(hpn);
[x1 x2]
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the PN Sequence Generator block reference page. The object properties correspond to the block parameters, except:

- The object does not have a property to select frame based outputs.
- The object does not have a property that corresponds to the **Sample time** parameter.

See Also

`comm.KasamiSequence | comm.GoldSequence`

comm.PNSequence.clone

Purpose Create PN sequence generator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `PNSequence` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.PNSequence.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `PNSequence` 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.

comm.PNSequence.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 PN sequence generator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `PNSequence` object, `H`.

Purpose

Generate a pseudo-noise (PN) sequence

Syntax

```
Y = step(H)
Y = step(H,MASK)
Y = step(H,RESET)
Y = step(H,MASK,RESET)
```

Description

`Y = step(H)` outputs a frame of the PN sequence in column vector `Y`. Specify the frame length with the `SamplesPerFrame` property. The PN sequence has a period of N = 2^n-1, where n is the degree of the generator polynomial that you specify in the `Polynomial` property.

`Y = step(H,MASK)` uses `MASK` as the shift value when you set the `MaskSource` property to 'Input port'. `MASK` must be a numeric, binary vector with length equal to the degree of the generator polynomial specified in the `Polynomial` property. Refer to the `Mask` property help for details of the mask calculation.

`Y = step(H,RESET)` uses `RESET` as the reset signal when you set the `ResetInputPort` property to true. The data type of the `RESET` input must be double precision or logical. `RESET` can be a scalar value or a column vector with length equal to the number of samples per frame specified in the `SamplesPerFrame` property. When the `RESET` input is a non zero scalar, the object resets to the initial conditions that you specify in the `InitialConditions` property and then generates a new output frame. A column vector `RESET` input allows multiple resets within an output frame. A non-zero value at the ith element of the vector will cause a reset at the ith output sample time. You can combine optional input arguments when you set their enabling properties. Optional inputs must be listed in the same order as the order of the enabling properties. For example,

```
Y = step(H,MASK,RESET)
```

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.

comm.PSKCarrierPhaseSynchronizer

Purpose

Recover carrier phase of baseband PSK signal

Description

The `PSKCarrierPhaseSynchronizer` object recovers the carrier phase of the input signal using the M-Power method. This feedforward method is not data aided but is clock aided. You can use this method for systems that use baseband phase shift keying (PSK) modulation. The method is also suitable for systems that use baseband quadrature amplitude modulation (QAM). However, the results are less accurate than those for comparable PSK systems. The alphabet size for the modulation requires an even integer.

Construction

`H = comm.PSKCarrierPhaseSynchronizer` creates a PSK carrier phase synchronizer System object, `H`. This object recovers the carrier phase of a baseband phase shift keying (PSK) modulated signal using the M-power method.

`H = comm.PSKCarrierPhaseSynchronizer(Name,Value)` creates a PSK carrier phase synchronizer 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`).

`H = comm.PSKCarrierPhaseSynchronizer(M,Name,Value)` creates a PSK carrier phase synchronizer object, `H`. This object has the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the modulation order of the input signal as an even, positive, real scalar value. Choose a data type of `single` or `double`. The default is `2`. This property is tunable.

ObservationInterval

Number of symbols where carrier phase assumed constant

Specify the observation interval as a real positive scalar integer value. Choose a data type of `single` or `double`. The default is `100`.

Methods

clone	Create PSK carrier phase synchronizer object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the PSK carrier phase synchronizer object
step	Recover baseband PSK signal's carrier phase

Examples

Recover carrier phase of a 16-PSK signal using M-power method.

```
M = 16;
phOffset = 10 *pi/180;  % in radians
numSamples = 100;
% Create PSK modulator System object
    hMod = comm.PSKModulator(M, phOffset, 'BitInput',false);
% Create PSK carrier phase synchronizer System object
    hSync = comm.PSKCarrierPhaseSynchronizer(M,...
                        'ObservationInterval',numSamples);
% Generate random data
    data = randi([0 M-1],numSamples,1);
% Modulate random data and add carrier phase
    modData = step(hMod, data);
% Recover the carrier phase
    [recSig phEst] = step(hSync, modData);
fprintf('The carrier phase is estimated to be %g degrees.\n', phEst);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PSK Phase Recovery block reference page. The object properties correspond to the block parameters.

See Also

`comm.CPMCarrierPhaseSynchronizer` | `comm.PSKModulator`

Purpose

Create PSK carrier phase synchronizer object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `PSKCarrierPhaseSynchronizer` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.PSKCarrierPhaseSynchronizer.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.PSKCarrierPhaseSynchronizer.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.PSKCarrierPhaseSynchronizer.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `PSKCarrierPhaseSynchronizer` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.PSKCarrierPhaseSynchronizer.reset

Purpose Reset states of the PSK carrier phase synchronizer object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `PSKCarrierPhaseSynchronizer` object, `H`.

Purpose

Recover baseband PSK signal's carrier phase

Syntax

```
[Y,PH] = step(H,X)
```

Description

`[Y,PH] = step(H,X)` recovers the carrier phase of the input signal, `X`, and returns the phase corrected signal, `Y`, and the carrier phase estimate (in degrees), `PH`. `X` must be a complex scalar or column vector input signal of data type single or double.

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

Demodulate using M-ary PSK method

Description

The `PSKDemodulator` object demodulates an input signal using the M-ary phase shift keying (M-PSK) method.

Construction

`H = comm.PSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the M-ary phase shift keying (M-PSK) method.

`H = comm.PSKDemodulator(Name,Value)` creates an M-PSK demodulator 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`).

`H = comm.PSKDemodulator(M,PHASE,Name,Value)` creates an M-PSK demodulator object, `H`. This object has the `ModulationOrder` property set to `M`, the `PhaseOffset` property set to `PHASE`, and the other specified properties set to the specified values. `M` and `PHASE` are value-only arguments. To specify a value-only argument, you must also specify all preceding value-only arguments. You can specify name-value pair arguments in any order.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is `8`.

PhaseOffset

Phase of zeroth point of constellation

Specify the phase offset of the zeroth point of the constellation, in radians, as a real scalar value. The default is `pi/8`.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. The default is `false`. When you set this property to `true`, the `step` method outputs a column vector of bit values. The length of this vector equals **log2**(`ModulationOrder`) times the number of demodulated symbols. When you set this property to `false`, the `step` method outputs a column vector with a length equal to the input data vector. This vector contains integer symbol values between `0` and `ModulationOrder-1`.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder`) bits to the corresponding symbol. Choose from `Binary | Gray | Custom`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the integer m, between $0 \le m \le$ `ModulationOrder`–1) maps to the complex value **exp**($j\times$ `PhaseOffset` + $j\times 2\times \pi \times m$/`ModulationOrder`). When you set this property to `Custom`, the object uses the signal constellation defined in the `CustomSymbolMapping` property.

CustomSymbolMapping

Custom constellation encoding

Specify a custom constellation symbol mapping vector. The default is `0:7`. This property requires a row or column vector with a size of `ModulationOrder`. This vector must have unique integer values in the range [`0`, `ModulationOrder`–1]. The values must be of data type `double`. The first element of this vector corresponds to the constellation point at an angle of `0 + PhaseOffset`, with subsequent elements running counterclockwise. The last element corresponds to the constellation point at an angle of $-\pi$ /`ModulationOrder + PhaseOffset`. This property applies when you set the `SymbolMapping` property to `Custom`.

DecisionMethod

Demodulation decision method

Specify the decision method the object uses as Hard decision | Log-likelihood ratio | Approximate log-likelihood ratio. The default is Hard decision. When you set the BitOutput property to false, the object always performs hard decision demodulation. This property applies when you set the BitOutput property to true.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of Property | Input port. The default is Property. This property applies when you set the BitOutput property to true and the DecisionMethod property to Log-likelihood ratio or Approximate log-likelihood ratio.

Variance

Noise variance

Specify the variance of the noise as a positive, real scalar value. The default is 1. If this value is very small (i.e., SNR is very high), log-likelihood ratio (LLR) computations may yield Inf or –Inf. This result occurs because the LLR algorithm computes the exponential of very large or very small numbers using finite-precision arithmetic. In such cases, use approximate LLR instead because the algorithm for that option does not compute exponentials. This property applies when you set the BitOutput property to true, the DecisionMethod property to Log-likelihood ratio, or Approximate log-likelihood ratio, and the VarianceSource property to Property. This property is tunable.

OutputDataType

Data type of output

Specify the output data type as Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32. The default is Full precision. This property applies when you set the BitOutput property to false. It also applies when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision`. In this second case, when the `OutputDataType` property is set to `Full precision`, the input data type is single- or double-precision, the output data has the same data type as the input. . When the input data is of a fixed-point type, the output data type behaves as if you had set the `OutputDataType` property to `Smallest unsigned integer`.

When you set `BitOutput` to `true` and the `DecisionMethod` property to `Hard Decision`, then `logical` data type becomes a valid option. If you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the output data has the same data type as the input. In this case, the data type must be single- or double-precision.

Fixed-Point Properties

DerotateFactorDataType

Data type of derotate factor

Specify the derotate factor data type as `Same word length as input | Custom`. The default is `Same word length as input`. This property applies when you set the `BitOutput` property to `false. It also applies` when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision`. The object uses the derotate factor in the computations only when the `ModulationOrder` property is `2`, `4`, or `8`. The `step` method input must also have a fixed-point type, and the `PhaseOffset` property must have a nontrivial value. For `ModulationOrder` = `2`, the phase offset is trivial if that value is a multiple of $\pi/2$. For `ModulationOrder` = `4`, the phase offset is trivial if that value is an even multiple of $\pi/4$. For `ModulationOrder` = `8`, there are no trivial phase offsets.

CustomDerotateFactorDataType

Fixed-point data type of derotate factor

Specify the derotate factor fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `DerotateFactorDataType` property to `Custom`. The word length must be a value between `2` and `128`.

Methods

clone	Create PSK demodulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Demodulate using M-ary PSK method

Examples

Modulate and demodulate a signal using 16-PSK modulation.

```
hMod = comm.PSKModulator(16, 'PhaseOffset',pi/16);
   hAWGN = comm.AWGNChannel('NoiseMethod', ...
         'Signal to noise ratio (SNR)','SNR',15);
   hDemod = comm.PSKDemodulator(16, 'PhaseOffset',pi/16);
   %Create an error rate calculator
   hError = comm.ErrorRate;
   for counter = 1:100
       % Transmit a 50-symbol frame
       data = randi([0 hMod.ModulationOrder-1],50,1);
       modSignal = step(hMod, data);
```

```
    noisySignal = step(hAWGN, modSignal);
    receivedData = step(hDemod, noisySignal);
    errorStats = step(hError, data, receivedData);
end
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.PSKModulator` | `comm.DPSKDemodulator`

comm.PSKDemodulator.clone

Purpose

Create PSK demodulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `PSKDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

Purpose Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.PSKDemodulator

Create a `comm.PSKDemodulator` System object, and then calculate its ideal signal constellation.

Create a `comm.PSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.PSKDemodulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.PSKDemodulator

Create a `comm.PSKDemodulator` System object, and then plot the ideal signal constellation.

Create a `comm.PSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.PSKDemodulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

comm.PSKDemodulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.PSKDemodulator.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `PSKDemodulator` 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)` Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.PSKDemodulator.step

Purpose Demodulate using M-ary PSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description `Y = step(H,X)` demodulates data, `X`, with the PSK demodulator System object, `H`, and returns `Y`. Input `X` must be a scalar or a column vector with double or single precision data type. If the value of the `ModulationOrder` property is less than or equal to `8` and you set `BitOutput` to `false`, or when you set the `DecisionMethod` property to `Hard Decision` and `BitOutput` to `true`, the object accepts an input with a signed integer data type or signed fixed point (fi objects). Depending on the `BitOutput` property value, output `Y`, can be integer or bit valued.

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the `BitOutput` property to true, the `DecisionMethod` property to `Approximate log-likelihood ratio` or `Log-likelihood ratio`, and the `VarianceSource` property to `Input port`. The data type of input `VAR` must be double or single precision.

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

Modulate using M-ary PSK method

Description

The `PSKModulator` object modulates using the M-ary phase shift keying method. The output is a baseband representation of the modulated signal. The M-ary number parameter, M, is the number of points in the signal constellation.

Construction

`H = comm.PSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the M-ary phase shift keying (M-PSK) method.

`H = comm.PSKModulator(Name,Value)` creates an M-PSK modulator 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`).

`H = comm.PSKModulator(M,PHASE,Name,Value)` creates an M-PSK modulator object, `H`. This object has the `ModulationOrder` property set to `M`, the `PhaseOffset` property set to `PHASE`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is `8`.

PhaseOffset

Phase of zeroth point of constellation

Specify the phase offset of the zeroth point of the constellation, in radians, as a real scalar value. The default is `pi/8`.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. When you set this property to `true`, the `step` method input must be a column vector of bit values. This vector must have a length that is an integer multiple of **log2**(ModulationOrder). This vector contains bit representations of integers between 0 and ModulationOrder-1. When you set the BitInput property to false, the step method input must be a column vector of numeric data type integer symbol values. These values must be between 0 and ModulationOrder-1. The default is false.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(ModulationOrder) input bits to the corresponding symbol as one of Binary | Gray | Custom. The default is Gray. When you set this property to Gray, the object uses a Gray-encoded signal constellation. When you set this property to Binary, the integer m, between 0 $\leq m \leq$ ModulationOrder–1) maps to the complex value **exp**($j\times$ PhaseOffset + $j\times 2\times \pi \times m$/ModulationOrder). When you set this property to Custom, the object uses the signal constellation defined in the CustomSymbolMapping property.

CustomSymbolMapping

Custom constellation encoding

Specify a custom constellation symbol mapping vector. This property requires a row or column vector of size ModulationOrder and must have unique integer values in the range [0, ModulationOrder–1]. The values must be of data type double. The first element of this vector corresponds to the constellation point at an angle of 0 + PhaseOffset, with subsequent elements running counterclockwise. The last element corresponds to the constellation point at an angle of $-\pi$ /ModulationOrder + PhaseOffset. This property applies when you set the SymbolMapping property to Custom. The default is 0:7.

OutputDataType

Data type of output

Specify the output data type as `double | single | Custom`. The default is `double`.

Fixed-Point Properties

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `OutputDataType` property to `Custom`.

Methods

clone	Create PSK modulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Modulate using M-ary PSK method

Examples

Modulate data using 16-PSK modulation, and visualize the data in a scatter plot.

```
% Create binary data for 24, 4-bit symbols
   data = randi([0 1],96,1);
```

```
% Create a 16-PSK modulator System object with bits as inputs and Gray-c
   hModulator = comm.PSKModulator(16,'BitInput',true);
% Change the phase offset to pi/16
   hModulator.PhaseOffset = pi/16;
% Modulate and plot the data
   modData = step(hModulator, data);
   constellation(hModulator)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

comm.PSKDemodulator | comm.QPSKModulator

Purpose Create PSK modulator object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `PSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.PSKModulator.constellation

Purpose Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.PSKModulator

Create a `comm.PSKModulator` System object, and then calculate its ideal signal constellation.

Create a `comm.PSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.PSKModulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.PSKModulator

Create a `comm.PSKModulator` System object, and then plot the ideal signal constellation.

Create a `comm.PSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.PSKModulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose

Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.PSKModulator.getNumOutputs

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `PSKModulator` 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.

comm.PSKModulator.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Modulate using M-ary PSK method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` modulates input data, `X`, with the PSK modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with numeric, logical, or fixed-point data types.

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.

---

comm.PSKTCMDemodulator

Purpose

Demodulate convolutionally encoded data mapped to M-ary PSK constellation

Description

The `PSKTCMDemodulator` object uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using a PSK signal constellation.

Construction

`H = comm.PSKTCMDemodulator` creates a trellis-coded, M-ary phase shift, keying (PSK TCM) demodulator System object, `H`. This object demodulates convolutionally encoded data that has been mapped to an M-PSK constellation.

`H = comm.PSKTCMDemodulator(Name,Value)` creates a PSK TCM demodulator 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`).

`H = comm.PSKTCMDemodulator(TRELLIS,Name,Value)` creates a PSK TCM demodulator System object, `H`. This object has the `TrellisStructure` property set to `TRELLIS` and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the `istrellis` function to check whether the trellis structure is valid. The default is the result of `poly2trellis([1 3], [1 0 0; 0 5 2])`.

TerminationMethod

Termination method of encoded frame

Specify the termination method as one of `Continuous | Truncated | Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object saves the internal state metric at the end of each frame. The next frame uses the same state metric. The object treats each traceback path independently. If the input signal contains only one symbol, use `Continuous` mode.

When you set this property to `Truncated`, the object treats each input vector independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state.

When you set property to `Terminated`, the object treats each input vector independently, and the traceback path always starts and ends in the all-zeros state.

TracebackDepth

Traceback depth for Viterbi decoder

Specify the scalar, integer number of trellis branches to construct each traceback path. The default is `21`. The traceback depth influences the decoding accuracy and delay. The decoding delay is the number of zero symbols that precede the first decoded symbol in the output.

When you set the `TerminationMethod` property to `Continuous`, the decoding delay consists of TracebackDepth zero symbols or TracebackDepth$\times K$ zero bits for a rate K/N convolutional code.

When you set the `TerminationMethod` property to `Truncated` or `Terminated`, no output delay occurs and the traceback depth must be less than or equal to the number of symbols in each input vector.

ResetInputPort

Enable demodulator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to initial conditions. This property applies when you set the `TerminationMethod` property to `Continuous`.

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation used to map the convolutionally encoded data as a positive, integer scalar value. The number of points must be `4`, `8`, or `16`. The default is `8`. The `ModulationOrder` property value must equal the number of possible input symbols to the convolutional decoder of the PSK TCM demodulator object. The `ModulationOrder` property must equal 2^N for a rate K/N convolutional code.

OutputDataType

Data type of output

Specify output data type as `logical | double`. The default is `double`.

Methods

clone	Create PSK TCM demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the PSK TCM demodulator object
step	Demodulate convolutionally encoded data mapped to M-ary PSK constellation

Examples

Modulate and demodulate data using 8-PSK TCM modulation.

```
% Define a trellis structure with 4-ary input symbols and 8-ary output
    t =  poly2trellis([5 4],[23 35 0; 0 5 13]);
    hMod = comm.PSKTCMModulator(t,'ModulationOrder', 8);
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)','SNR',6);
    hDemod = comm.PSKTCMDemodulator(t, 'ModulationOrder', 8, ...
                    'TracebackDepth',16);
% Create an error rate calculator with delay in bits equal to Tracebac
    hError = comm.ErrorRate('ReceiveDelay',...
                    hDemod.TracebackDepth*log2(t.numInputSymbols));
    for counter = 1:10
      % Transmit frames of 250 2-bit symbols
      data = randi([0 1],500,1);
      modSignal = step(hMod, data);
      noisySignal = step(hAWGN, modSignal);
      receivedData = step(hDemod, noisySignal);
      errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PSK TCM Decoder block reference page. The object properties correspond to the block parameters.

See Also

`comm.PSKTCMModulator` | `comm.GeneralQAMTCMDemodulator` | `comm.RectangularQAMTCMDemodulator` | `comm.ViterbiDecoder`

comm.PSKTCMDemodulator.clone

Purpose Create PSK TCM demodulator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `PSKTCMDemodulator` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.PSKTCMDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `PSKTCMDemodulator` 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.

comm.PSKTCMDemodulator.release

Purpose Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the PSK TCM demodulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `PSKTCMDemodulator` object, `H`.

comm.PSKTCMDemodulator.step

Purpose

Demodulate convolutionally encoded data mapped to M-ary PSK constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

`Y = step(H,X)` demodulates the PSK modulated input data, `X`, and uses the Viterbi algorithm to decode the resulting demodulated, convolutionally encoded bits. `X` must be a complex, double or single precision column vector. The `step` method outputs a demodulated, binary data column vector, `Y`. When the convolutional encoder represents a rate K/N code, the length of the output vector is $K \times L$, where L is the length of the input vector, `X`.

`Y = step(H,X,R)` resets the decoder to the all-zeros state when you input a reset signal, `R` that is non-zero. `R` must be a double precision or logical, scalar integer. This syntax applies 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.

Purpose

Convolutionally encode binary data and map using M-ary PSK constellation

Description

The `PSKTCMModulator` object implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal and then mapping the result to a PSK signal constellation.

Construction

`H = comm.PSKTCMModulator` creates a trellis-coded M-ary phase shift keying (PSK TCM) modulator System object, `H`. This object convolutionally encodes a binary input signal and maps the result to an M-PSK constellation.

`H = comm.PSKTCMModulator(Name,Value)` creates a PSK TCM encoder 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`).

`H = comm.PSKTCMModulator(TRELLIS,Name,Value)` creates a PSK TCM encoder object, `H`. This object has the `TrellisStructure` property set to `TRELLIS` and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the `istrellis` function to check whether a trellis structure is valid. The default is the result of `poly2trellis([1 3], [1 0 0; 0 5 2])`.

TerminationMethod

Termination method of encoded frame

Specify the termination method as one of `Continuous | Truncated | Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object retains the encoder states at the end of each input vector for use with the next input vector.

When you set this property to `Truncated`, the object treats each input vector independently. The encoder is reset to the all-zeros state at the start of each input vector.

When you set this property to `Terminated`, the object treats each input vector independently. However, for each input vector, the object uses extra bits to set the encoder to the all-zeros state at the end of the vector. For a rate K/N code, the `step` method outputs the vector with a length given by $y = N \times {}^{(L+S)}\!/_{K}$, where S = *constraintLength*–1 (or, in the case of multiple constraint lengths, S = **sum**(*constraintLength*(*i*)–1)). L indicates the length of the input to the `step` method.

ResetInputPort

Enable modulator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to initial conditions. This property applies when you set the `TerminationMethod` property to `Continuous`.

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation used to map the convolutionally encoded data as a positive integer scalar value equal to `4`, `8`, or `16`. The default is `8`. The value of the `ModulationOrder` property must equal the number of possible output symbols from the convolutional encoder of the PSK TCM modulator. Thus, the value for the `ModulationOrder` property must equal 2^N for a rate K/N convolutional code.

OutputDataType

Data type of output

Specify the output data type as one of `double | single`. The default is `double`.

Methods

clone	Create PSK TCM modulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the PSK TCM modulator object
step	Convolutionally encode binary data and map using M-ary PSK constellation

Examples

Modulate data using 8-PSK TCM modulation.

```
% Create binary data
    data = randi([0 1], 1000,1);
% Define a trellis structure with 4-ary input symbols and 8-ary output
    t =  poly2trellis([5 4],[23 35 0; 0 5 13]);
    hMod = comm.PSKTCMModulator(t,'ModulationOrder', 8);
% Modulate and plot the data
    modData = step(hMod, data);
    scatterplot(modData);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PSK TCM Decoder block reference page. The object properties correspond to the block parameters.

See Also

`comm.PSKTCMDemodulator` | `comm.GeneralQAMTCMModulator` | `comm.RectangularQAMTCMModulator` | `comm.ConvolutionalEncoder`

Purpose
Create PSK TCM modulator object with same property values

Syntax

```
C = clone(H)
```

Description
`C = clone(H)` creates a `PSKTCMModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.PSKTCMModulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose

Number of outputs from step method

Syntax

`N = getNumOutputs(H)`

Description

`N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.PSKTCMModulator.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `PSKTCMModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.PSKTCMModulator.reset

Purpose Reset states of the PSK TCM modulator object

Syntax

```
reset(H)
```

Description `reset(H)` resets the states of the `PSKTCMModulator` object, `H`.

Purpose

Convolutionally encode binary data and map using M-ary PSK constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

`Y = step(H,X)` convolutionally encodes and modulates the input binary data column vector, `X`, and returns the encoded and modulated data, `Y`. `X` must be of data type numeric, logical, or unsigned fixed point of word length `1` (fi object). When the convolutional encoder represents a rate K/N code, the length of the input vector, `X`, must be $K \times L$, for some positive integer L. The `step` method outputs a complex column vector, `Y`, of length L.

`Y = step(H,X,R)` resets the encoder of the PSK TCM modulator object to the all-zeros state when you input a reset signal, `R`, that is non-zero. `R` must be a double precision or logical scalar integer. This syntax applies 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.

Purpose

Demodulate using QPSK method

Description

The QPSKDemodulator object demodulates a signal that was modulated using the quaternary phase shift keying method. The input is a baseband representation of the modulated signal.

Construction

H = comm.QPSKDemodulator creates a demodulator System object, H. This object demodulates the input signal using the quadrature phase shift keying (QPSK) method.

H = comm.QPSKDemodulator(Name,Value) creates a QPSK demodulator 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).

H = comm.QPSKDemodulator(PHASE,Name,Value) creates a QPSK demodulator object, H. This object has the PhaseOffset property set to PHASE, and the other specified properties set to the specified values.

Properties

PhaseOffset

Phase of zeroth point in constellation

Specify the phase offset of the zeroth point in the constellation, in radians, as a real scalar value. The default is pi/4.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values.

When you set this property to true, the step method outputs a column vector of bit values with length equal to twice the number of demodulated symbols.

When you set this property to false, the step method outputs a column vector with length equal to the input data vector. This vector contains integer symbol values between 0 and 3. The default is false.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of 2 bits to the corresponding symbol as one of `Binary | Gray`. The default is `Gray`.

When you set this property to `Gray`, the object uses a Gray-encoded signal constellation.

When you set this property to `Binary`, the integer `m`, between $0 \leq m \leq 3$ maps to the complex value **exp**($j\times$ `PhaseOffset` + $j\times 2\pi \times {}^{m}\!/_{4}$).

DecisionMethod

Demodulation decision method

Specify the decision method the object uses as `Hard decision | Log-likelihood ratio | Approximate log-likelihood ratio`. The default is `Hard decision`.

When you set the `BitOutput` property to `false`, the object always performs hard decision demodulation. This property applies when you set the `BitOutput` property to `true`.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of `Property | Input port`. The default is `Property`. This property applies when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`.

Variance

Noise variance

Specify the variance of the noise as a positive, real scalar value. The default is `1`. If this value is very small (i.e., SNR is very high), log-likelihood ratio (LLR) computations may yield `Inf` or `-Inf`. This result occurs because the LLR algorithm computes the exponential of very large or very small numbers using finite-precision arithmetic. In such cases, use approximate LLR is because that option's algorithm does not compute exponentials.

This property applies when you set the `BitOutput` property to `true`, the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, and the `VarianceSource` property to `Property`.This property is tunable.

OutputDataType

Data type of output

Specify the output data type as `Full precision` | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`. The default is `Full precision`.

This property applies when you set the `BitOutput` property to `false`. The property also applies when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision`. In this second case, when the `OutputDataType` property is set to `Full precision`, and the input data type is single or double precision, the output data has the same as that of the input.

When the input data is of a fixed-point type, the output data type behaves as if you had set the `OutputDataType` property to `Smallest unsigned integer`.

When you set `BitOutput` to `true` and the `DecisionMethod` property to `Hard Decision`, then `logical` data type becomes a valid option.

When you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the output data type is the same as that of the input. In this case, that data type can only be single or double precision.

Fixed-Point Properties

DerotateFactorDataType

Data type of derotate factor

Specify derotate factor data type as one of `Same word length as input | Custom`. The default is `Same word length as input`.

This property applies when you set the `BitOutput` property to `false`. The property also applies when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision`. The object uses the derotate factor in the computations only when the step method input is a fixed-point type and the `PhaseOffset` property has a value that is not an even multiple of pi/4.

CustomDerotateFactorDataType

Fixed-point data type of derotate factor

Specify the derotate factor fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `DerotateFactorDataType` property to `Custom`.

Methods

clone	Create QPSK demodulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Demodulate using QPSK method

Examples

Modulate and demodulate a signal using QPSK modulation.

```
hMod = comm.QPSKModulator('PhaseOffset',pi/4);
hAWGN = comm.AWGNChannel('NoiseMethod',...
                     'Signal to noise ratio (SNR)','SNR',15);
hDemod = comm.QPSKDemodulator('PhaseOffset',pi/4);
%Create an error rate calculator
hError = comm.ErrorRate;
for counter = 1:100
    % Transmit a 50-symbol frame
    data = randi([0 3],50,1);
    modSignal = step(hMod, data);
    noisySignal = step(hAWGN, modSignal);
    receivedData = step(hDemod, noisySignal);
    errorStats = step(hError, data, receivedData);
end
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the QPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

comm.QPSKModulator | comm.PSKDemodulator

Purpose Create QPSK demodulator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `QPSKDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.QPSKDemodulator.constellation

Purpose

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.QPSKDemodulator

Create a `comm.QPSKDemodulator` System object, and then calculate its ideal signal constellation.

Create a `comm.QPSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.QPSKDemodulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.QPSKDemodulator

Create a `comm.QPSKDemodulator` System object, and then plot the ideal signal constellation.

Create a `comm.QPSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.QPSKDemodulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.QPSKDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `QPSKDemodulator` 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.

comm.QPSKDemodulator.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 Demodulate using QPSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

`Y = step(H,X)` demodulates input data, `X`, with the QPSK demodulator System object, `H`, and returns `Y`. Input `X` must be a scalar or a column vector with double or single precision data type. When you set the `BitOutput` property to `false`, or when you set the `DecisionMethod` property to `Hard decision` and the `BitOutput` property to `true`, the data type of the input can also be signed integer, or signed fixed point (fi objects). Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the `BitOutput` property to `true`, the `DecisionMethod` property to `Approximate log-likelihood ratio`or `Log-likelihood ratio`, and the `VarianceSource` property to `Input port`. The data type of input `VAR` must be double or single precision.

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 Modulate using QPSK method

Description The `QPSKModulator` object modulates using the quaternary phase shift keying method. The output is a baseband representation of the modulated signal.

Construction `H = comm.QPSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the quadrature phase shift keying (QPSK) method.

`H = comm.QPSKModulator(Name,Value)` creates a QPSK modulator 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`).

`H = comm.QPSKModulator(PHASE,Name,Value)` creates a QPSK modulator object, `H`. This object has the `PhaseOffset` property set to `PHASE` and the other specified properties set to the specified values.

Properties

PhaseOffset

Phase of zeroth point in constellation

Specify the phase offset of the zeroth point in the constellation, in radians, as a real scalar value. The default is pi/4.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`. When you set this property to `true`, the `step` method input must be a column vector of bit values. This vector must have a length that is an integer multiple of `2`. This vector contains bit representations of integers between `0` and `3`. When you set this property to `false`, the step method input must be a column vector of integer symbol values between 0 and 3.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or a group of two input bits to the corresponding symbol as one of `Binary | Gray`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the input integer m, between $0 \le m \le 3$, maps to the complex value $\mathbf{exp}(j \times \texttt{PhaseOffset} + j \times 2 \times \pi \times m/4)$.

OutputDataType

Data type of output

Specify the output data type as one of `double | single | Custom`. The default is `double`.

Fixed-Point Properties

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `OutputDataType` property to `Custom`.

Methods

clone	Create QPSK modulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Modulate using QPSK method

Examples

Modulate data using QPSK, and visualize the data in a scatter plot.

```
% Create binary data for 48, 2-bit symbols
      data = randi([0 1],96,1);
% Create a QPSK modulator System object with bits as inputs and Gray-coc
      hModulator = comm.QPSKModulator('BitInput',true);
% Change the phase offset to pi/16
      hModulator.PhaseOffset = pi/16;
% Modulate and plot the data
      modData = step(hModulator, data);
      scatterplot(modData)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the QPSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

comm.QPSKDemodulator | comm.PSKModulator

Purpose Create QPSK modulator object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `QPSKModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.QPSKModulator.constellation

Purpose

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.QPSKModulator

Create a `comm.QPSKModulator` System object, and then calculate its ideal signal constellation.

Create a `comm.QPSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.QPSKModulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.QPSKModulator

Create a `comm.QPSKModulator` System object, and then plot the ideal signal constellation.

Create a `comm.QPSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.PSKModulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose

Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.QPSKModulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `QPSKModulator` 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.

comm.QPSKModulator.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Modulate using QPSK method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` modulates input data, `X`, with the QPSK modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with numeric, logical, or fixed-point data types.

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.

---

comm.RectangularQAMDemodulator

Purpose

Demodulate using rectangular QAM method

Description

The `RectangularQAMDemodulator` object demodulates a signal that was modulated using quadrature amplitude modulation with a constellation on a rectangular lattice.

Construction

`H = comm.RectangularQAMDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the rectangular quadrature amplitude modulation (QAM) method.

`H = comm.RectangularQAMDemodulator(Name,Value)` creates a rectangular QAM demodulator 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`).

`H = comm.RectangularQAMDemodulator(M,Name,Value)` creates a rectangular QAM demodulator object, `H`. This object has the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as scalar value with a positive, integer power of two. The default is `16`.

PhaseOffset

Phase offset of constellation

Specify the phase offset of the signal constellation, in radians, as a real scalar value. The default is `0`.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. When you set this property to true the step method outputs a column vector of bit values whose length equals **log2**(ModulationOrder) times the number of demodulated symbols. When you set this property to false, the step method outputs a column vector with a length equal to the input data vector. This vector contains integer symbol values between 0 and ModulationOrder-1. The default is false.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(ModulationOrder) bits to the corresponding symbol as one of Binary | Gray | Custom. The default is Gray. When you set this property to Gray, the object uses a Gray-coded signal constellation. When you set this property to Binary, the object uses a natural binary-coded constellation. When you set this property to Custom, the object uses the signal constellation defined in the CustomSymbolMapping property.

CustomSymbolMapping

Custom constellation encoding

Specify a custom constellation symbol mapping vector. The default is 0:15. This property is a row or column vector with a size of ModulationOrder and with unique integer values in the range [0, ModulationOrder-1]. The values must be of data type double. The first element of this vector corresponds to the top-leftmost point of the constellation, with subsequent elements running down column-wise, from left to right. The last element corresponds to the bottom-rightmost point. This property applies when you set the SymbolMapping property to Custom.

NormalizationMethod

Constellation normalization method

Specify the method used to normalize the signal constellation as Minimum distance between symbols | Average power | Peak power. The default is Minimum distance between symbols.

MinimumDistance

Minimum distance between symbols

Specify the distance between two nearest constellation points as a positive, real, numeric scalar value. The default is `2`. This property applies when you set the `NormalizationMethod` property to `Minimum distance between symbols`.

AveragePower

Average power of constellation

Specify the average power of the symbols in the constellation as a positive, real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` property to `Average power`.

PeakPower

Peak power of constellation

Specify the maximum power of the symbols in the constellation as a positive, real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` property to `Peak power`.

DecisionMethod

Demodulation decision method

Specify the decision method the object uses as `Hard decision` | `Log-likelihood ratio` | `Approximate log-likelihood ratio`. The default is `Hard decision`. When you set the `BitOutput` property to `false` the object always performs hard-decision demodulation. This property applies when you set the `BitOutput` property to `true`.

VarianceSource

Source of noise variance

Specify the source of the noise variance as `Property` | `Input port`. The default is `Property`. This property applies when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`.

Variance

Noise variance

Specify the variance of the noise as a positive, real scalar value. The default is `1`. If this value is very small (i.e., SNR is very high), log-likelihood ratio (LLR) computations may yield Inf or -Inf. This result occurs because the LLR algorithm computes the exponential of very large or very small numbers using finite-precision arithmetic. In such cases, using approximate LLR is recommended because its algorithm does not compute exponentials. This property applies when you set the `BitOutput` property to `true`, the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, and the `VarianceSource` property to `Property`. This property is tunable.

OutputDataType

Data type of output

Specify the output data type as `Full precision` | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`. The default is `Full precision`.

This property applies only when you set the `BitOutput` property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision`. In this case, when the `OutputDataType` property is set to `Full precision`, and the input data type is single- or double-precision, the output data has the same data type as the input.

When the input data is of a fixed-point type, the output data type behaves as if you had set the `OutputDataType` property to `Smallest unsigned integer`.

When you set the BitOutput property to true and the DecisionMethod property to Hard Decision, then logical data type becomes a valid option.

When you set the BitOutput property to true and the DecisionMethod property to Log-likelihood ratio or Approximate log-likelihood ratio, the output data type is the same as that of the input. In this case, that data type can only be single- or double-precision.

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".

DerotateFactorDataType

Data type of derotate factor

Specify the derotate factor data type as Same word length as input | Custom. The default is Same word length as input. This property applies when you set the BitOutput property to false, or when you set the BitOutput property to true and the DecisionMethod property to Hard decision. The object uses the derotate factor in the computations only when the step method input is of a fixed-point type and the `PhaseOffset` property has a value that is not a multiple of $\pi/2$.

CustomDerotateFactorDataType

Fixed-point data type of derotate factor

Specify the derotate factor fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `DerotateFactorDataType` property to `Custom`.

DenormalizationFactorDataType

Data type of denormalization factor

Specify the denormalization factor data type as `Same word length as input | Custom`. The default is `Same word length as input`. This property applies when you set the `BitOutput` property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision`.

CustomDenormalizationFactorDataType

Fixed-point data type of denormalization factor

Specify the denormalization factor fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `DenormalizationFactorDataType` property to `Custom`.

ProductDataType

Data type of product

Specify the product data type as `Full precision | Custom`. The default is `Full precision`. This property applies when you set the `BitOutput` property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision`.

CustomProductDataType

Fixed-point data type of product

Specify the product fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32)`. This property applies when you set the `ProductDataType` property to `Custom`.

ProductRoundingMethod

Rounding of fixed-point numeric value of product

Specify the product rounding method as `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`. This property applies when the object is not in a full precision configuration, when you set the `BitOutput` property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision`.

ProductOverflowAction

Action when fixed-point numeric value of product overflows

Specify the product overflow action as `Wrap` | `Saturate`. The default is `Wrap`. This property applies when the object is not in a full precision configuration, when you set the `BitOutput` property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision`.

SumDataType

Data type of sum

Specify the sum data type as `Full precision` | `Same as product` | `Custom`. The default is `Full precision`. This property applies when you set the `FullPrecisionOverride` property to false, when you set the `BitOutput` property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` property to `Hard decision`.

CustomSumDataType

Fixed-point data type of sum

Specify the sum fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32)`. This property applies when you set the `FullPrecisionOverride` property to false or when you set the `SumDataType` property `Custom`.

Methods

clone	Create rectangular QAM demodulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Demodulate using rectangular QAM method

Examples

Modulate and demodulate data using rectangular 8-QAM.

```
   hMod = comm.RectangularQAMModulator(8);
   hAWGN = comm.AWGNChannel('NoiseMethod',...
               'Signal to noise ratio (SNR)', 'SNR', 15, 'SignalPower
   hDemod = comm.RectangularQAMDemodulator(8);
% Create an error rate calculator
   hError = comm.ErrorRate;
   for counter = 1:100
% Transmit a 50-symbol frame
       data = randi([0 hMod.ModulationOrder-1],50,1);
```

```
    modData = step(hMod, data);
    receivedSignal = step(hAWGN, modData);
    receivedData = step(hDemod, receivedSignal);
    errorStats = step(hError, data, receivedData);
end
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Rectangular QAM Demodulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.RectangularQAMModulator` | `comm.GeneralQAMDemodulator`

Purpose

Create rectangular QAM demodulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `RectangularQAMDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.RectangularQAMDemodulator.constellation

Purpose

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.RectangularQAMDemodulator

Create a `comm.RectangularQAMDemodulator` System object, and then calculate its ideal signal constellation.

Create a `comm.RectangularQAMDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.RectangularQAMDemodulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.RectangularQAMDemodulator

Create a `comm.RectangularQAMDemodulator` System object, and then plot the ideal signal constellation.

Create a `comm.RectangularQAMDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.RectangularQAMDemodulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose

Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.RectangularQAMDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `RectangularQAMDemodulator` 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.

comm.RectangularQAMDemodulator.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Demodulate using rectangular QAM method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

`Y = step(H,X)` demodulates the input data, `X`, with the rectangular QAM demodulator System object, `H`, and returns, `Y`. Input `X` must be a scalar or a column vector with double or single precision data type. When `ModulationOrder` is an even power of two and you set the `BitOutput` property to `false` or, when you set the `DecisionMethod` to `Hard decision` and the `BitOutput` property to `true`, the data type of the input can also be signed integer, or signed fixed point (fi objects). Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the `BitOutput` property to `true`, the `DecisionMethod` property to `Approximate log-likelihood ratio`or `Log-likelihood ratio`, and the `VarianceSource` property to `Input port`. The data type of input `VAR` must be double or single precision.

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.

---

comm.RectangularQAMModulator

Purpose

Modulate using rectangular QAM method

Description

The `RectangularQAMModulator` object modulates using M-ary quadrature amplitude modulation with a constellation on a rectangular lattice. The output is a baseband representation of the modulated signal. This block accepts a scalar or column vector input signal.

Construction

`H = comm.RectangularQAMModulator` creates a modulator object, `H`. This object modulates the input using the rectangular quadrature amplitude modulation (QAM) method.

`H = comm.RectangularQAMModulator(Name,Value)` creates a rectangular QAM modulator 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`).

`H = comm.RectangularQAMModulator(M,Name,Value)` creates a rectangular QAM modulator object, `H`. This object has the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as scalar value that is a positive integer power of two. The default is `16`.

PhaseOffset

Phase offset of constellation

Specify the phase offset of the signal constellation, in radians, as a real scalar value. The default is `0`.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`. When you set this property to `true`, the `step` method input requires a column vector of bit values. The length of this vector must an integer multiple of **log2**(ModulationOrder). This vector contains bit representations of integers between 0 and ModulationOrder–1. When you set this property to false, the step method input must be a column vector of integer symbol values between 0 and ModulationOrder–1.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(ModulationOrder) input bits to the corresponding symbol as Binary | Gray | Custom. The default is Gray. When you set this property to Gray, the System object uses a Gray-coded signal constellation. When you set this property to Binary, the object uses a natural binary-coded constellation. When you set this property to Custom, the object uses the signal constellation defined in the CustomSymbolMapping property.

CustomSymbolMapping

Custom constellation encoding

Specify a custom constellation symbol mapping vector. The default is 0:15. This property is a row or column vector with a size of ModulationOrder. This vector has unique integer values in the range [0, ModulationOrder–1]. These values must be of data type double. The first element of this vector corresponds to the top-leftmost point of the constellation, with subsequent elements running down column-wise, from left to right. The last element corresponds to the bottom-rightmost point. This property applies when you set the SymbolMapping property to Custom.

NormalizationMethod

Constellation normalization method

Specify the method used to normalize the signal constellation as Minimum distance between symbols | Average power | Peak power. The default is Minimum distance between symbols.

MinimumDistance

Minimum distance between symbols

Specify the distance between two nearest constellation points as a positive, real, numeric scalar value. The default is `2`. This property applies when you set the `NormalizationMethod` property to `Minimum distance between symbols`.

AveragePower

Average power of constellation

Specify the average power of the symbols in the constellation as a positive, real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` property to `Average power`.

PeakPower

Peak power of constellation

Specify the maximum power of the symbols in the constellation as a positive real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` property to `Peak power`.

OutputDataType

Data type of output

Specify the output data type as `double | single | Custom`. The default is `double`.

Fixed-Point Properties

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `OutputDataType` property to `Custom`.

Methods

clone	Create rectangular QAM modulator object with same property values
constellation	Calculate or plot ideal signal constellation
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Modulate using rectangular QAM method

Examples

Modulate data using rectangular QAM modulation, and visualize the data in a scatter plot.

```
% Create binary data for 32, 3-bit symbols
    data = randi([0 1],96,1);
% Create a rectangular 8-QAM modulator System object with bits as inpu
    hModulator = comm.RectangularQAMModulator(8,'BitInput',true);
% Rotate the constellation by pi/4 radians
    hModulator.PhaseOffset = pi/4;
% Modulate and plot the data
    modData = step(hModulator, data);
    constellation(hModulator)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Rectangular QAM Modulator Baseband block reference page. The object properties correspond to the block parameters.

See Also

`comm.RectangularQAMDemodulator` | `comm.GeneralQAMModulator`

Purpose

Create rectangular QAM modulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `RectangularQAMModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.RectangularQAMModulator.constellation

Purpose

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.RectangularQAMModulator

Create a `comm.RectangularQAMModulator` System object, and then calculate its ideal signal constellation.

Create a `comm.RectangularQAMModulator` System object by entering the following at the MATLAB command line:

```
h = comm.RectangularQAMModulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.RectangularQAMModulator

Create a `comm.RectangularQAMModulator` System object, and then plot the ideal signal constellation.

Create a `comm.RectangularQAMModulator` System object by entering the following at the MATLAB command line:

```
h = comm.RectangularQAMModulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

Purpose Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.RectangularQAMModulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `RectangularQAMModulator` 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.

comm.RectangularQAMModulator.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Modulate using rectangular QAM method

Syntax

```
Y = step(H,X)
```

Description

`Y = step(H,X)` modulates input data, `X`, with the rectangular QAM modulator object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with numeric, logical, or fixed-point data types.

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.

---

comm.RectangularQAMTCMDemodulator

Purpose

Demodulate convolutionally encoded data mapped to rectangular QAM constellation

Description

The `RectangularQAMTCMDemodulator` object uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using a rectangular QAM signal constellation.

Construction

`H = comm.RectangularQAMTCMDemodulator` creates a trellis-coded, rectangular, quadrature amplitude (QAM TCM) demodulator System object, `H`. This object demodulates convolutionally encoded data that has been mapped to a rectangular QAM constellation.

`H = comm.RectangularQAMTCMDemodulator(Name,Value)` creates a rectangular, QAM TCM, demodulator 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`).

`H = comm.RectangularQAMTCMDemodulator(TRELLIS,Name,Value)` creates a rectangular QAM TCM demodulator object, `H`. This object has the `TrellisStructure` property set to `TRELLIS`, and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the `istrellis` function to check whether a structure is a valid trellis. The default is the result of `poly2trellis([3 1 1], [ 5 2 0 0; 0 0 1 0; 0 0 0 1])`.

TerminationMethod

Termination method of encoded frame

Specify the termination method as `Continuous | Truncated | Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object saves the internal state metric at the end of each frame. The next frame uses the same state metric. The object treats each traceback path independently. If the input signal contains only one symbol, you should use `Continuous` mode.

When you set this property to `Truncated`, the object treats each input vector independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state.

When you set this property to `Terminated`, the object treats each input vector independently, and the traceback path always starts and ends in the all-zeros state.

TracebackDepth

Traceback depth for Viterbi decoder

Specify the scalar, integer number of trellis branches to construct each traceback path. The default is `21`. The Traceback depth parameter influences the decoding accuracy and delay. The decoding delay is the number of zero symbols that precede the first decoded symbol in the output.

When you set the `TerminationMethod` property to `Continuous`, the decoding delay consists of TracebackDepth zero symbols or TracebackDepth$\times K$ zero bits for a rate K/N convolutional code.

When you set the `TerminationMethod` property to `Truncated` or `Terminated`, no output delay occurs and the traceback depth must be less than or equal to the number of symbols in each input vector.

ResetInputPort

Enable demodulator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to initial conditions. This property applies when you set the `TerminationMethod` property to `Continuous`.

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation used to map the convolutionally encoded data as a positive, integer scalar value. The number of points must be `4`, `8`, `16`, `32`, or `64`. The default is `16`. The `ModulationOrder` property value must equal the number of possible input symbols to the convolutional decoder of the rectangular QAM TCM demodulator object. The `ModulationOrder` must equal 2^N for a rate K/N convolutional code.

OutputDataType

Data type of output

Specify output data type as `logical | double`. The default is `double`.

Methods

clone	Create rectangular QAM TCM demodulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the rectangular QAM TCM demodulator object
step	Demodulate convolutionally encoded data mapped to rectangular QAM constellation

Examples

Modulate and demodulate data using rectangular 16-QAM TCM modulation.

```
    hMod = comm.RectangularQAMTCMModulator;
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)',...
                    'SNR',5, 'SignalPower', 10);
    hDemod = comm.RectangularQAMTCMDemodulator('TracebackDepth',16);
% Create an error rate calculator with delay in bits equal to Traceba
    delay = hDemod.TracebackDepth* ...
            log2(hDemod.TrellisStructure.numInputSymbols);
    hErrorCalc = comm.ErrorRate('ReceiveDelay', delay);
    for counter = 1:10
      % Transmit frames of 200 3-bit symbols
      data = randi([0 1],600,1);
      modSignal = step(hMod, data);
      noisySignal = step(hAWGN, modSignal);
      receivedData = step(hDemod, noisySignal);
      errorStats = step(hErrorCalc, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Rectangular QAM TCM Decoder block reference page. The object properties correspond to the block parameters.

See Also

`comm.RectangularQAMTCMModulator` | `comm.GeneralQAMTCMDemodulator` | `comm.ViterbiDecoder`

comm.RectangularQAMTCMDemodulator.clone

Purpose

Create rectangular QAM TCM demodulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `RectangularQAMTCMDemodulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.RectangularQAMTCMDemodulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.RectangularQAMTCMDemodulator.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

`TF = isLocked(H)`

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `RectangularQAMTCMDemodulator` 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.

comm.RectangularQAMTCMDemodulator.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the rectangular QAM TCM demodulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `RectangularQAMTCMDemodulator` object, `H`.

comm.RectangularQAMTCMDemodulator.step

Purpose

Demodulate convolutionally encoded data mapped to rectangular QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

`Y = step(H,X)` demodulates the rectangular QAM modulated input data, `X`, and uses the Viterbi algorithm to decode the resulting demodulated, convolutionally encoded bits. `X` must be a complex, double or single precision column vector. The step method outputs a demodulated, binary data column vector, `Y`. When the convolutional encoder represents a rate K/N code, the length of the output vector is K*L, where L is the length of the input vector, `X`.

`Y = step(H,X,R)` resets the decoder to the all-zeros state when you input a reset signal, `R` that is non-zero. `R` must be a double precision or logical, scalar integer. This syntax applies 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.

Purpose

Convolutionally encode binary data and map using rectangular QAM constellation

Description

The `RectangularQAMTCMModulator` object implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal and mapping the result to a rectangular QAM signal constellation.

Construction

`H = comm.RectangularQAMTCMModulator` creates a trellis-coded, rectangular, quadrature amplitude (QAM TCM) System object, `H`. This object convolutionally encodes a binary input signal and maps the result to a rectangular QAM constellation.

`H = comm.RectangularQAMTCMModulator(Name,Value)` creates a rectangular QAM TCM modulator 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`).

`H = comm.RectangularQAMTCMModulator(TRELLIS,Name,Value)` creates a rectangular QAM TCM modulator object, `H`. This object has the `TrellisStructure` property set to `TRELLIS` and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the `istrellis` function to check whether a structure is a valid trellis. The default is the result of `poly2trellis([3 1 1], [ 5 2 0 0; 0 0 1 0; 0 0 0 1])`.

TerminationMethod

Termination method of encoded frame

Specify the termination method as `Continuous | Truncated | Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object retains the encoder states at the end of each input vector for use with the next input vector.

When you set this property to `Truncated`, the object treats each input vector independently. The encoder is reset to the all-zeros state at the start of each input vector.

When you set this property to `Terminated`, the object treats each input vector independently. For each input vector, the object uses extra bits to set the encoder to the all-zeros state at the end of the vector. For a rate *K/N* code, the `step` method outputs the vector with a length given by $y = N \times {}^{(L+S)}\!/_{K}$, where S = *constraintLength*–1 (or, in the case of multiple constraint lengths, S = **sum**(*constraintLength*(*i*)–1)). L is the length of the input to the `step` method.

ResetInputPort

Enable modulator reset input

Set this property to true to enable an additional input to the step method. The default is `false`. When you set the reset input to the step method to a nonzero value, the object resets the encoder to the all-zeros state. This property applies when you set the `TerminationMethod` property to `Continuous`.

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation used to map the convolutionally encoded data as a positive integer scalar value equal to `4`, `8`, `16`, `32`, or `64`. The default is `16`. The value of the `ModulationOrder` property must equal the number of possible output symbols from the convolutional encoder of the QAM TCM modulator. Thus, the value for the `ModulationOrder` property must equal 2^N for a rate *K/N* convolutional code.

OutputDataType

Data type of output

Specify the output data type as one of `double | single`. The default is `double`.

Methods

clone	Create rectangular QAM TCM modulator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the rectangular QAM TCM modulator object
step	Convolutionally encode binary data and map using rectangular QAM constellation

Examples

Modulate data using rectangular 16-QAM TCM modulation.

```
% Create binary data.
    data = randi([0 1], 3000,1);
    hMod = comm.RectangularQAMTCMModulator;
% Modulate and plot the data.
    modData = step(hMod, data);
    scatterplot(modData);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Rectangular QAM TCM Encoder block reference page. The object properties correspond to the block parameters.

See Also

`comm.RectangularQAMTCMDemodulator` | `comm.GeneralQAMTCMModulator` | `comm.ConvolutionalEncoder`

Purpose

Create rectangular QAM TCM modulator object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `RectangularQAMTCMModulator` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.RectangularQAMTCMModulator.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.RectangularQAMTCMModulator.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.RectangularQAMTCMModulator.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `RectangularQAMTCMModulator` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.RectangularQAMTCMModulator.reset

Purpose Reset states of the rectangular QAM TCM modulator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `RectangularQAMTCMModulator` object, `H`.

Purpose

Convolutionally encode binary data and map using rectangular QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

`Y = step(H,X)` convolutionally encodes and modulates the input data numeric or logical column vector `X`, and returns the encoded and modulated data, `Y`. `X` must be of data type numeric, logical, or unsigned fixed point of word length `1` (fi object). When the convolutional encoder represents a rate K/N code, the length of the input vector, `X`, must be $K \times L$, for some positive integer L. The `step` method outputs a complex column vector, Y, of length L.

`Y = step(H,X,R)` resets the encoder of the rectangular QAM TCM modulator object to the all-zeros state when you input a non-zero reset signal, `R`. `R` must be a double precision or logical, scalar integer. This syntax applies 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.

Purpose

Decode data using Reed-Solomon decoder

Description

The `RSDecoder` object recovers a message vector from a Reed-Solomon codeword vector. For proper decoding, the property values for this object should match those in the corresponding RS Encoder object.

Construction

`H = comm.RSDecoder` creates a block decoder System object, `H`. This object performs Reed-Solomon (RS) decoding.

`H = comm.RSDecoder(Name,Value)` creates an RS decoder 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`).

`H = comm.RSDecoder(N,K,Name,Value)` creates an RS decoder object, `H`. This object has the `CodewordLength` property set to `N`, the `MessageLength` property set to `N`, and the other specified properties set to the specified values.

Properties

BitInput

Assume input is bits

Specify whether the input comprises bits or integers. The default is `false`.

When you set this property to `false`, the `step` method input data value must be a numeric, column vector of integers. The length of this vector must equal an integer multiple of (`CodewordLength` – number of punctures). You specify the number of punctures with the `PuncturePatternSource` and `PuncturePattern` properties. The `CodewordLength` property stores the codeword length value. The decoded data output result is a column vector of integers. The length of this vector equals an integer multiple of the message length you specify in the `MessageLength` property. Each symbol that forms the input codewords and output message is an integer between `0` and 2^M–1. These integers correspond to an element of the finite Galois field `GF`(2^M). *M* is the degree of the primitive polynomial that you specify with the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties.

When you set this property to `true`, the input encoded data value must be a numeric column vector of bits. The length equal to an integer multiple of (`CodewordLength` – number of punctures)$\times M$. You specify the number of punctures with `PuncturePatternSource` and `PuncturePattern` properties. The decoded data output result is a column vector of bits. The length equals an integer multiple of `MessageLength`$\times M$. A group of M bits represents an integer between 0 and 2^M–1 that belongs to the finite Galois field `GF`(2^M). M is the degree of the primitive polynomial that you specify with the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties.

When you set `BitInput` to `false` and `ErasuresInputPort` to `true`, the erasures input, `ERASURES`, must be set to a length equal to the encoded data input vector. Values of 1 in the `ERASURES` vector correspond to erased symbols in the same position as the input codewords, and values of `0` correspond to nonerased symbols.

When you set this property to `true` and `ErasuresInputPort` to `true`, `ERASURES`, requires a length of $1/M$ times the length of the input encoded data vector. M corresponds to the degree of the primitive polynomial. Values of `1` in the `ERASURES` vector correspond to erased symbols and values of `0` correspond to nonerased symbols. In this case, a symbol corresponds to M bits.

CodewordLength

Codeword length

Specify the codeword length of the RS code as a double-precision, positive, integer scalar value. The default is `7`.

If you set the `PrimitivePolynomialSource` property to `Auto`, `CodewordLength` must be in the range 3 << `CodewordLength` $\leq 2^{16}$–1.

If you set the PrimitivePolynomialSource property to Property, CodewordLength must be in the range 3 ≤ CodewordLength ≤ 2^M–1. M is the degree of the primitive polynomial that you specify with the PrimitivePolynomialSource and PrimitivePolynomial properties. M must be in the range $3 \leq M \leq 16$. The difference (CodewordLength – MessageLength) must be an even integer.

For a full-length RS code the value of the CodewordLength property requires the form 2^M–1. If this property is less than 2^M–1, the object assumes a shortened RS code.

MessageLength

Message length

Specify the message length as a double-precision, positive, integer scalar value. The default is 3. The difference (CodewordLength –MessageLength) must be an even integer.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as Auto | Property. The default is Auto.

When you set this property to Auto, the object uses a primitive polynomial of degree M = ceil(**log2**(CodewordLength+1)), which is the result of fliplr(de2bi(primpoly(M))).

When you set this property to Property you specify a polynomial using the PrimitivePolynomial property.

PrimitivePolynomial

Primitive polynomial

Specify the primitive polynomial that defines the finite field GF(2^M) corresponding to the integers that form messages and codewords. The default is the result of fliplr(de2bi(primpoly(*3*))), which is [1 0 1 1] or the polynomial x^3+x+1. You must set this property to a double-precision, binary, row vector that represents a primitive polynomial over GF(2) of degree M in descending order of powers. If CodewordLength is less than 2^M–1, the object uses a shortened RS code. This property applies when you set the PrimitivePolynomialSource property to Property.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as Auto | Property. The default is Auto.

When you set this property to Auto, the object automatically chooses the generator polynomial. The object calculates the generator polynomial based on the value of the PrimitivePolynomialSource property.

When you set the PrimitivePolynomialSource property to Auto the object calculates the generator polynomial as rsgenpoly(CodewordLength+SL,MessageLength+SL).

When you set the PrimitivePolynomialSource property to Property, the object calculates generator polynomial as rsgenpoly(CodewordLength+SL,MessageLength+SL, PrimitivePolynomial). In both cases, SL = (2^M–1) –CodewordLength is the shortened length, and M is the degree of the primitive polynomial that you specify with the PrimitivePolynomialSource and PrimitivePolynomial properties. When you set this property to Property, you can specify a generator polynomial using this property.

GeneratorPolynomial

Generator polynomial

Specify the generator polynomial for the RS code as a double-precision, integer row vector or as a Galois field row vector whose entries are in the range from 0 to 2^M-1 and represent a generator polynomial in descending order of powers. The length of the generator polynomial must be

CodewordLength-MessageLength+1. This property applies when you set the GeneratorPolynomialSource property to Property.

The default is the result of rsgenpoly(7,3,[],[],'double'), which corresponds to [1 3 1 2 3].

When you use this object to generate code, you must set the generator polynomial to a double-precision, integer row vector.

CheckGeneratorPolynomial

Enable generator polynomial checking

Set this property to true to perform a generator polynomial check. The default is true. This check verifies that $\times^{\texttt{CodewordLength}}$ + 1 is divisible by the generator polynomial you specify in the GeneratorPolynomial property.

For larger codes, disabling the check accelerates processing time. You should perform the check at least once before setting this property to false. This property applies when you set the GeneratorPolynomialSource property to Property.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as None | Property. The default is None. If you set this property to None then the object does not apply puncturing to the code. If you set this property to Property then the object punctures the code based on a puncture pattern vector specified in the PuncturePattern property.

PuncturePattern

Puncture pattern vector

Specify the pattern used to puncture the encoded data as a double-precision, binary column vector of length (CodewordLength-MessageLength). The default is [ones(2,1); zeros(2,1)]. Zeros in the puncture pattern vector indicate the position of the parity symbols that are punctured or excluded from each codeword. This property applies when you set the `PuncturePatternSource` property to `Property`.

ErasuresInputPort

Enable erasures input

Set this property to `true` to specify a vector of erasures as an input to the `step` method. The default is `false`. The erasures input must be a double-precision or logical binary column vector that indicates which symbols of the input codewords to erase.

When you set `BitInput` to `true`, the erasures vector length must equal 1/*M* times the length of the input encoded data vector, where *M* corresponds to the degree of the primitive polynomial. Values of `1` in the erasures vector correspond to erased symbols in the same position of the bit-packed input codewords. Values of 0 correspond to nonerased symbols.

When you set `BitInput` to `false`, the erasures vector length must equal the input encoded data vector. Values of `1` in the erasures vector correspond to erased symbols in the same position of the input codewords. Values of `0` correspond to nonerased symbols.

When this property is set to `false` the object assumes no erasures.

NumCorrectedErrorsOutputPort

Enable number of corrected errors output

Set this property to `true` to obtain the number of corrected errors as an output to the `step` method. The default is `true`. A nonnegative value in the *i*-th element of the error output vector, denotes the number of corrected errors in the *i*-th input codeword. A value of `-1` in the *i*-th element of the error output vector indicates that a decoding error occurred for that codeword. A decoding error occurs when an input codeword has more errors than the error correction capability of the RS code.

OutputDataType

Data type of output

Specify the output data type as `Same as input | double | logical`. The default is `Same as input`. This property applies when you set the `BitInput` property to `true`.

Methods

clone	Create RS decoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Decode data using a Reed-Solomon decoder

Examples

Transmit an RS-encoded, 8-DPSK-modulated symbol stream through an AWGN channel. Then, demodulate, decode, and count errors.

```
hEnc = comm.RSEncoder;
hMod = comm.DPSKModulator('BitInput',false);
hChan = comm.AWGNChannel(...
            'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
hDemod = comm.DPSKDemodulator('BitOutput',false);
hDec = comm.RSDecoder;
hError = comm.ErrorRate('ComputationDelay',3);

for counter = 1:20
  data = randi([0 7], 30, 1);
  encodedData = step(hEnc, data);
  modSignal = step(hMod, encodedData);
  receivedSignal = step(hChan, modSignal);
```

```
  demodSignal = step(hDemod, receivedSignal);
  receivedSymbols = step(hDec, demodSignal);
  errorStats = step(hError, data, receivedSymbols);
end

fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Binary-Output RS Decoder and Integer-Output RS Decoder block reference pages. The object properties correspond to the block parameters, except:

> The `BitInput` property allows you to select between the Binary-Output RS Decoder and Integer-Output RS Decoder algorithms.

See Also

`comm.RSEncoder` | `comm.BCHDecoder`

comm.RSDecoder.clone

Purpose Create RS decoder object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `RSDecoder` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.RSDecoder.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `RSDecoder` 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.

comm.RSDecoder.release

Purpose

Allow property value and input characteristics changes

Syntax

`release(H)`

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 data using a Reed-Solomon decoder

Syntax

```
[Y,ERR] = step(H,X)
Y = step(H,X)
Y = step(H,X,ERASURES)
```

Description

`[Y,ERR] = step(H,X)` decodes the encoded input data, `X`, into the output vector `Y` and returns the number of corrected errors in output vector `ERR`. The value of the `BitInput` property determines whether `X` is a vector of integers or bits with a numeric, logical, or fixed-point data type. The `PuncturePatternSource` and `PuncturePattern` properties affect the expected length of `X`. The `MessageLength` property affects the length of `Y`. This syntax applies when you set the `NumCorrectedErrorsOutputPort` property to true.

`Y = step(H,X)` decodes the encoded data, `X`, into the output vector `Y`. This syntax applies when you set the `NumCorrectedErrorsOutputPort` property to false.

`Y = step(H,X,ERASURES)` uses the binary column input vector, `ERASURES`, to erase the symbols of the input codewords. The elements in `ERASURES` must be of data type double or logical. Values of 1 in the `ERASURES` vector correspond to erased symbols, and values of 0 correspond to non-erased symbols. This syntax applies when you set the `ErasuresInputPort` property to true. See the `ErasuresInputPort` property help for more information.

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.

---

comm.RSEncoder

Purpose

Encode data using Reed-Solomon encoder

Description

The RSEncoder object creates a Reed-Solomon code with message and codeword lengths you specify.

Construction

H = comm.RSEncoder creates a block encoder System object, H. This object performs Reed-Solomon (RS) encoding.

H = comm.RSEncoder(Name,Value) creates an RS encoder 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).

H = comm.RSEncoder(N,K,Name,Value) creates an RS encoder object, H. This object has the CodewordLength property set to N, the MessageLength property set to K, and the other specified properties set to the specified values.

Properties

BitInput

Assume input is bits

Specify whether the input comprises bits or integers. The default is false.

When you set this property to false, the step method input data value must be a numeric, column vector of integers. The length equals an integer multiple of the message length value stored in the MessageLength property. Each group of MessageLength input elements represents one message word the object will encode.

The step method outputs an encoded data output vector. The output result is a column vector of integers. The length is an integer multiple of (CodewordLength – number of punctures). You specify the number of punctures with the PuncturePatternSource and PuncturePattern properties. Each symbol that forms the input message and output codewords is an integer between 0 and 2^M–1. These integers correspond to an element of the finite Galois field GF(2^M). M is the degree of the primitive polynomial that you specify with the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties.

When you set this property to `true`, the input value must be a numeric, column vector of bits with an integer multiple of `MessageLength`$\times M$ bits. Each group of `MessageLength`$\times M$ input bits represents one message word the object will encode. The encoded data output result is a column vector of bits. The length of this vector equals an integer multiple of (`CodewordLength` – number of punctures)$\times M$. You specify the number of punctures with the `PuncturePatternSource` and `PuncturePattern` properties. A group of M bits represents an integer between `0` and 2^M–1 that belongs to the finite Galois field $GF(2^M)$. M is the degree of the primitive polynomial that you specify with the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties.

CodewordLength

Codeword length

Specify the codeword length of the RS code as a double-precision, positive, integer scalar value. The default is `7`.

If you set the `PrimitivePolynomialSource` property to `Auto`, `CodewordLength` must be in the range 3 < `CodewordLength` ≤ 2^{16}–1.

When you set the `PrimitivePolynomialSource` property to `Property`, `CodewordLength` must be in the range 3 ≤ `CodewordLength` ≤ 2^M–1. M is the degree of the primitive polynomial that you specify with the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties. M must be in the range $3 \leq M \leq 16$. The difference (`CodewordLength` –`MessageLength`) must be an even integer. For a full-length RS code, the value of this property requires the form 2^M–1.

If the value of this property is less than 2^M–1, the object assumes a shortened RS code.

MessageLength

Message length

Specify the message length as a double-precision, positive integer scalar value. The default is `3`. The difference (`CodewordLength` – `MessageLength`) must be an even integer.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as `Auto | Property`. The default is `Auto`.

When you set this property to `Auto`, the object uses a primitive polynomial of degree M = `ceil`(**log2**(`CodewordLength`+1)), which is the result of `fliplr`(`de2bi`(`primpoly`(M))).

When you set this property to `Property`, you can specify a polynomial using the `PrimitivePolynomial` property.

PrimitivePolynomial

Primitive polynomial

Specify the primitive polynomial that defines the finite field `GF`(2^M) corresponding to the integers that form messages and codewords. You must set this property to a double-precision, binary row vector that represents a primitive polynomial over `GF`(2) of degree M in descending order of powers.

If `CodewordLength` is less than 2^M–1, the object uses a shortened RS code. The default is the result of `fliplr`(`de2bi`(`primpoly`(3))), which is [1 0 1 1] or the polynomial $x^M + x + 1$.

This property applies when you set the `PrimitivePolynomialSource` property to `Property`.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as `Auto | Property`. The default is `Auto`.

When you set this property to `Auto`, the object automatically chooses the generator polynomial. The object calculates the generator polynomial based on the value of the `PrimitivePolynomialSource` property.

When you set the `PrimitivePolynomialSource` property to `Auto` the object calculates the generator polynomial as `rsgenpoly`(`CodewordLength`+SL,`MessageLength`+SL).

When you set the `PrimitivePolynomialSource` property to `Property`, the object computes generator polynomial as `rsgenpoly`(`CodewordLength`+SL,`MessageLength`+SL, `PrimitivePolynomial`). In both cases, SL = (2^M–1)–`CodewordLength` is the shortened length, and M is the degree of the primitive polynomial that you specify with the `PrimitivePolynomialSource` and PrimitivePolynomial properties.

When you set this property to `Property`, you can specify a generator polynomial using the `GeneratorPolynomial` property.

GeneratorPolynomial

Generator polynomial

Specify the generator polynomial for the RS code as a double-precision, integer row vector or as a Galois row vector whose entries are in the range from `0` to `2`^{M}`-1`and represent a generator polynomial in descending order of powers. Each coefficient is an element of Galois field `GF`(2^M) represented in integer format. The default is the result of `rsgenpoly`(7,3,[],[],'double'), which evaluates to a `GF`(2^3) array with elements `[1 3 1 2 3]`. This property applies when you set the `GeneratorPolynomialSource` property to `Property`.

CheckGeneratorPolynomial

Enable generator polynomial checking

Set this property to `true` to perform a generator polynomial check. The default is `true`. This check verifies that $\times^{\texttt{CodewordLength}}$

+ 1 is divisible by the generator polynomial specified in the GeneratorPolynomial property. For larger codes, disabling the check speeds up processing. You should perform the check at least once before setting this property to false. This property applies when you set the GeneratorPolynomialSource property to Property.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as None | Property. The default is None. If you set this property to None then the object does not apply puncturing to the code. If you set this property to Property then the object punctures the code based on a puncture pattern vector specified in the PuncturePattern property.

PuncturePattern

Puncture pattern vector

Specify the pattern used to puncture the encoded data as a double-precision, binary column vector with a length of (CodewordLength–MessageLength). The default is [ones(2,1); zeros(2,1)]. Zeros in the puncture pattern vector indicate the position of the parity symbols that are punctured or excluded from each codeword. This property applies when you set the PuncturePatternSource property to Property.

OutputDataType

Data type of output

Specify the output data type as Same as input | double | logical. The default is Same as input. This property applies when you set the BitInput property to true.

Methods

clone	Create RS encoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Encode data using a Reed-Solomon encoder

Examples

Transmit an RS-encoded, 8-DPSK-modulated symbol stream through an AWGN channel. Then, demodulate, decode, and count errors.

```
hEnc = comm.RSEncoder;
hMod = comm.DPSKModulator('BitInput',false);
hChan = comm.AWGNChannel(...
            'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
hDemod = comm.DPSKDemodulator('BitOutput',false);
hDec = comm.RSDecoder;
hError = comm.ErrorRate('ComputationDelay',3);

for counter = 1:20
  data = randi([0 7], 30, 1);
  encodedData = step(hEnc, data);
  modSignal = step(hMod, encodedData);
  receivedSignal = step(hChan, modSignal);
  demodSignal = step(hDemod, receivedSignal);
  receivedSymbols = step(hDec, demodSignal);
  errorStats = step(hError, data, receivedSymbols);
end
```

```
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Binary-Input RS Encoder and Integer-Input RS Encoder block reference pages. The object properties correspond to the block parameters, except for:

The `BitInput` property allows you to select between the Binary-Input RS Encoder and Integer-Input RS Encoder algorithms.

See Also

`comm.RSDecoder` | `comm.BCHEncoder`

Purpose

Create RS encoder object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `RSEncoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.RSEncoder.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

comm.RSEncoder.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description `TF = isLocked(H)` returns the locked status, `TF` of the `RSEncoder` 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)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.RSEncoder.step

Purpose Encode data using a Reed-Solomon encoder

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` encodes the numeric column input data vector, `X`, and returns the encoded data, `Y`. The value of the `BitInput` property determines whether `X` is a vector of integers or bits with a numeric, logical, or fixed-point data type. The `MessageLength` property affects the expected length of `X`. The `PuncturePatternSource` and `PuncturePattern` properties affect the length of `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

Scramble input signal

Description

The `Scrambler` object scrambles a scalar or column vector input signal.

Construction

`H = comm.Scrambler` creates a scrambler System object, `H`. This object scrambles the input data using a linear feedback shift register that you specify with the Polynomial property.

`H = comm.Scrambler(Name,Value)` creates a scrambler 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`).

`H = comm.Scrambler(N,POLY,COND,Name,Value)` creates a scrambler object, `H`. This object has the `CalculationBase` property set to `N`, the `Polynomial` property set to `POLY`, the `InitialConditions` property set to `COND`, and the other specified properties set to the specified values.

Properties

CalculationBase

Range of input data

Specify calculation base as a positive, integer, scalar value. Set the calculation base property to one greater than the number of input values. The `step` method input and output integers are in the range [0, CalculationBase–1]. The default is `4`.

Polynomial

Linear feedback shift register connections

Specify the polynomial that determines the shift register feedback connections. The default is `[1 1 1 0 1]`. You can the generator polynomial as a numeric, binary vector that lists the coefficients of the polynomial in order of ascending powers of z^{-1}, where $p(z^{-1}) = 1 + p1z^{-1} + p2z^{-2} + ...$ is the generator polynomial. The first and last elements must be `1`. Alternatively, you can specify the generator polynomial as a numeric vector. This vector must contain the exponents of z^{-1} for the nonzero terms of the polynomial, in order of ascending powers of z^{-1}. In this case, the first vector element must be `0`. For example, both `[1 0 0 0 0 0 1 0 1]` and `[0 -6 -8]` specify the same polynomial $p(z^{-1}) = 1 + z^{-6} + z^{-8}$.

InitialConditions

Initial values of linear feedback shift register

Specify the initial values of the linear feedback shift register as an integer row vector with values in [0 CalculationBase–1]. The default is `[0 1 2 3]`. The length of this property vector must equal the order of the `Polynomial` property vector.

Methods

clone	Create scrambler object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of scrambler object
step	Scramble input signal

Examples

Scramble and descramble random data with values in the range [0 7].

```
% Create scrambler and descrambler objects with calculation base of 8
N = 8;
hSCR = comm.Scrambler(N, [1 0 1 1 0 1 0 1],...
       [0 3 2 2 5 1 7]);
hDSCR = comm.Descrambler(N, [1 0 1 1 0 1 0 1],...
       [0 3 2 2 5 1 7]);
```

```
for counter = 1:10
  data = randi([0 N-1], 4, 1);
  scrData = step(hSCR, data);
  deScrData = step(hDSCR, scrData);
  [data, scrData, deScrData]
end
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Scrambler block reference page. The object properties correspond to the block parameters.

See Also

comm.Descrambler | comm.PNSequence

comm.Scrambler.clone

Purpose Create scrambler object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `Scrambler` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.Scrambler.getNumInputs

Purpose Number of expected inputs to step method

Syntax

```
N = getNumInputs(H)
```

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.Scrambler.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `Scrambler` 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.

comm.Scrambler.release

Purpose

Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description

`release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 scrambler object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `Scrambler` object, `H`.

comm.Scrambler.step

Purpose Scramble input signal

Syntax `Y = step(H,X)`

Description `Y = step(H,X)` scrambles input data, `X`, and returns the result in `Y`. `X` must be a double precision, logical, or integer column vector. The output `Y` is same data type and length as the input vector.

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 input using sphere decoder

Description

The `Sphere Decoder` System object decodes the symbols sent over `Nt` antennas using the sphere decoding algorithm.

Construction

`H = comm.SphereDecoder` creates a System object, `H`. This object uses the sphere decoding algorithm to find the maximum-likelihood solution for a set of received symbols over a MIMO channel with N_t transmit antennas and N_r receive antennas.

`H = comm.SphereDecoder(Name,Value)` creates a sphere decoder object, `H`, with the specified property name set to the specified value. Name must appear inside single quotes (”). You can specify several name-value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

`H = comm.SphereDecoder(CONSTELLATION,BITTABLE)` creates a sphere decoder object, `H`, with the Constellation property set to `CONSTELLATION`, and the `BitTable` property set to `BITTABLE`.

Properties

Constellation

Signal constellation per transmit antenna

Specify the constellation as a complex column vector containing the constellation points to which the transmitted bits are mapped. The default setting is a QPSK constellation with an average power of `1`. The length of the vector must be a power of two. The object assumes that each transmit antenna uses the same constellation.

BitTable

Bit mapping used for each constellation point.

Specify the bit mapping for the symbols that the `Constellation` property specifies as a numerical matrix. The default is `[0 0; 0 1; 1 0; 1 1]`, which matches the default `Constellation` property value.

The matrix size must be [ConstellationLength bitsPerSymbol]. ConstellationLength represents the length of the Constellation property. bitsPerSymbol represents the number of bits that each symbol encodes.

InitialRadius

Initial search radius of the decoding algorithm.

Specify the initial search radius for the decoding algorithm as either Infinity | ZF Solution. The default is Infinity.

When you set this property to Infinity, the object sets the initial search radius to Inf.

When you set this property to ZF Solution, the object sets the initial search radius to the zero-forcing solution. This calculation uses the pseudo-inverse of the input channel when decoding. Large constellations and/or antenna counts can benefit from the initial reduction in the search radius. In most cases, however, the extra computation of the ZF Solution will not provide a benefit.

DecisionType

Specify the decoding decision method as either Soft | Hard. The default is Soft.

When you set this property to Soft, the decoder outputs log-likelihood ratios (LLRs), or soft bits.

When you set this property to Hard, the decoder converts the soft LLRs to bits. The hard-decision output logical array follows the mapping of a zero for a negative LLR and one for all other values.

Methods

clone	Create object with same property values
isLocked	Locked status for input attributes and nontunable properties

release	Allow property value and input characteristics changes
step	Decode received symbols using sphere decoding algorithm

Examples

Decode using a sphere decoder

Modulate a set of bits using 16-QAM constellation. Transmit the signal as two parallel streams over a MIMO channel. Then, decode using a sphere decoder, with perfect channel knowledge.

Define the modulation order, number of bits to transmit, and the noise variance. Variance is directly related to noise power. The greater the variance value, the smaller the signal-to-noise ratio.

```
M = 16;
nBits = 1e3*log2(M);
noiseVariance = 1e-2;
symMap = [11 10 14 15 9 8 12 13 1 0 4 5 3 2 6 7];
```

Create a Rectangular QAM modulator System object with the `BitInput` property set to `true`, the `NormalizationMethod` set to `Average Power`, the `SymbolMapping` set to `Binary`, and `CustomSymbolMapping` set to `symMap`.

```
hMod = comm.RectangularQAMModulator('BitInput', true, ...
       'ModulationOrder', M, 'NormalizationMethod', 'Average power', ...
       'SymbolMapping', 'Custom', 'CustomSymbolMapping', symMap);
```

Convert the decimal value of the symbol map to binary bits using the left bit as the most significant bit (msb).

```
BitTable = de2bi(symMap, log2(M), 'left-msb');
```

Create a MIMO Channel System object with the `RandomStream` property set to `mt19937ar with seed` and `PathGainsOutputPort` set to `true` and a default configuration of 2-by-2.

```
hMIMO = comm.MIMOChannel('RandomStream', 'mt19937ar with seed',...
       'PathGainsOutputPort', true);
```

Create an AWGN Channel System object with the `NoiseMethod` property set to `Variance`, `VarianceSource` set to `Property`, and `Variance` set to `noiseVariance`.

```
 hAWGN = comm.AWGNChannel('NoiseMethod', 'Variance',...
        'VarianceSource', 'Property', 'Variance', 'noiseVariance');
```

Create a Sphere Decoder System object that processes bits using hard-decision decoding.

```
hSpDec = comm.SphereDecoder('Constellation', constellation(hMod),...
        'BitTable', BitTable, 'DecisionType', 'Hard');
```

Create an error rate System object.

```
hBER = comm.ErrorRate;
```

Generate a random data stream.

```
data = randi([0 1], nBits, 1);
```

Modulate the data by calling the `step` method of the Rectangular QAM Modulator System object, `hMod`.

```
 yMod = step(hMod, data);
```

Split the modulated data stream into two, and then transmit over a 2-by-2 MIMO fading channel.

```
yTx  = reshape(yMod, [], 2);
[yFad, yPG] = step(hMIMO, yTx);
```

Add noise to the received signal by calling the `step` method of the AWGN System object, `hAWGN`.

```
 yRec = step(hAWGN, yFad);
```

Decode the received signal.

```
rxBits = step(hSpDec, yRec, squeeze(yPG));
```

Calculate and then display the bit error rate results.

```
ber = step(hBER, data, double(rxBits(:)));
disp(ber(1));
```

LTE PDSCH Processing Using Sphere Decoding

See this example which simulates LTE PDSCH spatial multiplexing with sphere decoding.

Algorithm

This object implements a soft-output max-log APP MIMO detector by means of a soft-output Schnorr-Euchner sphere decoder (SESD), implemented as single tree search (STS) tree traversal. The algorithm assumes the same constellation and bit table on all of the transmit antennas. Given as inputs, the received symbol vector and the estimated channel matrix, the algorithm outputs the log-likelihood ratios (LLRs) of the transmitted bits.

The algorithm assumes a MIMO system model with N_t transmit antennas and N_r receive antennas where N_t symbols are simultaneously sent, which is express as:

$$y = Hs + n$$

where the received symbols y are a function of the transmitted symbol vector s, the MIMO channel matrix, H, and the thermal noise, n.

The goal of the MIMO detector is to find the maximum-likelihood (ML) solution, for which it holds that

$$\hat{s}_{ML} = \underset{s \in o}{\arg\min} \|y - Hs\|^2$$

where O is the complex-valued constellation from which the N_t elements of s are chosen.

Soft detection additionally delivers, for each bit, estimates on how reliable the estimate is. For each of the sent bits, denoted as $x_{j,b}$ (the b-th bit of the j-th symbol), the reliability of the estimate is calculated by means of the log-likelihood ratio (LLR), which is denoted as L and is calculated as using the max-log approximation:

$$L\left(x_{i,j}\right) = \underbrace{\min_{s \in x_{j,b}^{(0)}} \left\| y - Hs \right\|^2}_{\lambda^{ML}} - \underbrace{\min_{s \in x_{j,b}^{(1)}} \left\| y - Hs \right\|^2}_{\lambda_{j,b}^{\overline{ML}}}$$

where $x_{j,b}^{(0)}$ and $x_{j,b}^{(1)}$ are the disjoint sets of vector symbols that have the b-th bit in the label of the j-th scalar symbol equal to 0 and 1, respectively. The symbol λ denotes the distance calculated as norm squared. The two terms can be expressed as the difference of:

1. The distance to the ML solution $\hat{s}_{ML}$, denoted as λ^{ML}.
2. The distance $\lambda_{j,b}^{\overline{ML}}$ to the counter-hypothesis, which denotes the binary complement of the b-th bit in the binary label of the j-th entry of $\hat{s}_{ML}$, i.e., the minimum of the symbol set $x_{j,b}^{\left(x_{j,b}^{\overline{ML}}\right)}$, which contains all of the possible vectors for which the b-th bit of the j-th entry is flipped compared to the same entry of $\hat{s}_{ML}$.

Thus, depending on whether $x_{j,b}^{\left(x_{j,b}^{ML}\right)}$ is zero or one, the LLR for the bit $x_{j,b}$ is expressed as

$$L(x_{j,b}) = \begin{cases} \lambda^{ML} - \lambda_{j,b}^{\overline{ML}} & , x_{j,b}^{ML} = 0 \\ \lambda_{j,b}^{\overline{ML}} - \lambda^{ML} & , x_{j,b}^{ML} = 1 \end{cases}$$

The design of a decoder thus aims at efficiently finding $\hat{s}_{ML}$, λ^{ML}, and $\lambda_{j,b}^{\overline{ML}}$.

This search can be converted into a tree search by means of the sphere decoding algorithms. To this end, the channel matrix is decomposed into $H = QR$ by means of a QR decomposition. Left-multiplying y by Q^H, the problem can be reformulated as

$$\lambda^{ML} = \underset{s \in o}{\arg\min} \left\| \bar{y} - Rs \right\|^2$$

$$\lambda_{j,b}^{\overline{ML}} = \underset{s \in x_{j,b}^{\left(x_{j,b}^{\overline{ML}}\right)}}{\arg\min} \left\| \bar{y} - Rs \right\|^2$$

from which the triangular structure of R can be exploited to arrange a tree structure where each of the leaf nodes corresponds to a possible s vector and the partial distances to the nodes in the tree can be calculated cumulatively adding to the partial distance of the parent node.

In the STS algorithm, the λ^{ML} and $\lambda_{j,b}^{\overline{ML}}$ metrics are searched concurrently. The main idea is to have a list containing the metric λ^{ML}, along with the corresponding bit sequence x^{ML} and the metrics $x_{j,b}^{\left(x_{j,b}^{ML}\right)}$ of all counter-hypotheses. Then, we search the sub-tree originating from a given node only if the result can lead to an update of either λ^{ML} or $\lambda_{j,b}^{\overline{ML}}$.

The STS algorithm flow can be summarized as:

1. If when reaching a leaf node, a new ML hypothesis is found $\left(d(x) < \lambda^{ML}\right)$, all $\lambda_{j,b}^{\overline{ML}}$ for which $x_{j,b} = x_{j,b}^{\overline{ML}}$ are set to λ^{ML} which now turns into a valued counter-hypothesis. Then, λ^{ML} is set to the current distance d(x).

2. If the current partial distance d(x) satisfies $d(x) \geq \lambda^{ML}$, only the counter-hypotheses have to be checked. For all *j* and *b* for which $\left(d(x) < \lambda^{ML}\right)$ and $x_{j,b} = x_{j,b}^{\overline{ML}}$ the decoder updates $\lambda_{j,b}^{\overline{ML}}$ to be $d(x)$.

3. A sub-tree is pruned if the partial distance of the node is bigger than the current $\lambda_{j,b}^{\overline{ML}}$ which may be affected when traversing the subtree.

4. The algorithm finalizes once all of the tree nodes have been visited once or pruned.

Selected Bibliography

[1] Studer, C., M. Wenk, A. Burg, and H. Bölcskei. "Soft-output MIMO detection algorithms: Performance and implementation aspects" *Proceedings of the 40th Asilomar Conference on signals, Systems, and Computers*, October 2006.

[2] Cho, Y. S., et.al. "MIMO-OFDM Wireless communications with MATLAB," IEEE Press, 2011.

[3] Hochwald, B.M., S. ten Brink. “Achieving near-capacity on a multiple-antenna channel”, IEEE Transactions on Communications, Vol. 51, No. 3, Mar 2003, pp. 389-399.

[4] Agrell, E., T. Eriksson, A. Vardy, K. Zeger. “Closest point search in lattices”, IEEE Transactions on Information Theory, Vol. 48, No. 8, Aug 2002, pp. 2201-2214.

See Also

`comm.LTEMIMOChannel` | `comm.MIMOChannel` | `comm.OSTBCCombiner`

Purpose

Create `SphereDecoder` object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `SphereDecoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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.

comm.SphereDecoder.isLocked

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `SphereDecoder` 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 or hardware connections) and allows all properties and input characteristics to be changed.

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.

---

comm.SphereDecoder.step

Purpose

Decode received symbols using sphere decoding algorithm

Syntax

`Y = step(H, RXSYMBOLS, CHAN)`

Description

`Y = step(H, RXSYMBOLS, CHAN)` decodes the received symbols, RXSYMBOLS, using the sphere decoding algorithm. The algorithm can be employed to decode Ns channel realizations in one call, where in each channel realization, Nr symbols are received.

The inputs are:

`RXSYMBOLS`: a [Ns Nr] complex double matrix containing the received symbols.

`CHAN`: a [Ns Nt Nr] or [1 Nt Nr] complex double matrix representing the fading channel coefficients of the flat-fading MIMO channel. For the [Ns Nt Nr] case, the object applies each channel matrix to each Nr symbol set. For the block fading case, i.e., when the size of `CHAN` is [1 Nt Nr], the same channel is applied to all of the received symbols.

The output `Y`, which depends on the setting of the `DecisionType` property, is a double matrix containing the Log-Likelihood Ratios (LLRs) of the decoded bits or the bits themselves. For both cases, the size of the output is [Ns*bitsPerSymbol Nt], where `bitsPerSymbol` represents the number of bits per transmitted symbol, as determined by the `BitTable` 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

Add receiver thermal noise

Description

The `ThermalNoise` object simulates the effects of thermal noise on a complex, baseband signal.

Construction

`H = comm.ThermalNoise` creates a receiver thermal noise System object, `H`. This object adds thermal noise to the complex, baseband input signal.

`H = comm.ThermalNoise(Name,Value)` creates a receiver thermal noise 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

NoiseMethod

Method to specify noise level

Select the method to specify the noise level as `Noise temperature | Noise figure | Noise factor`. The default is `Noise temperature`.

NoiseTemperature

Noise temperature

Specify the noise temperature in degrees Kelvin as a numeric, nonnegative, real scalar value. The default is `290`. This property applies when you set the `NoiseMethod` property to `Noise temperature`.

NoiseFigure

Noise figure

Specify the noise figure in decibels relative to a noise temperature of 290 K. You must set this property to a numeric, nonnegative, real scalar value. This property applies when you set the `NoiseMethod` property to `Noise figure`. The default is `3.01` dB, which corresponds to a noise temperature of $290 \times (10^{(\mathrm{NoiseFigure}/10)} - 1)$. This value approximates 290 K.

NoiseFactor

Noise factor

Specify the noise factor as a factor relative to a noise temperature of 290 K. You must set this property to a numeric, real scalar value greater than or equal to 1. This property applies when you set the `NoiseMethod` property to `Noise factor`. The default is 2, which corresponds to a noise temperature of 290× (`NoiseFactor`–1) = 290 K.

SampleRate

Sample time

Specify the sample rate of the input samples in Hz as a numeric, real, positive scalar. The default is `1`. The object computes the variance of the noise added to the input signal as (`kT*SampleRate`). The value k is Boltzmann's constant and T is the noise temperature specified explicitly or implicitly via one of the noise methods.

Methods

clone	Create thermal noise object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Add receiver thermal noise

Examples

Add thermal noise with a noise temperature of 290 K to QPSK data.

```
    hTNoise = comm.ThermalNoise('NoiseTemperature',290);
% Create a modulator and obtain complex baseband signal
    hMod = comm.QPSKModulator;
    data = randi([0 3],32,1);
    modData = step(hMod,data);
% Add noise to signal
    noisyData = step(hTNoise,modData);
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Receiver Thermal Noise block reference page. The object properties correspond to the block parameters, except:

- This object uses the MATLAB default random stream to generate random numbers. The block uses a random number generator based on the V5 RANDN (Ziggurat) algorithm. The block also uses an initial seed, set with the **Initial seed** parameter to initialize the random number generator. Ever time the system that contains the block runs, the block generates the same sequence of random numbers. To generate reproducible numbers using this object, you can reset the MATLAB default random stream using the following code.

  ```
  reset(RandStream.getGlobalStream)
  ```

  For more information, see help for `RandStream`.

- The object provides a `SampleRate` property, which needs to be specified. The block senses the sample time of the signal and therefore does not have a corresponding parameter.

See Also

`comm.AWGNChannel`

comm.ThermalNoise.clone

Purpose Create thermal noise object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `ThermalNoise` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.ThermalNoise.getNumInputs

Purpose Number of expected inputs to step method

Syntax `N = getNumInputs(H)`

Description `N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.ThermalNoise.getNumOutputs

Purpose Number of outputs from step method

Syntax

```
N = getNumOutputs(H)
```

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose

Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `ThermalNoise` 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.

comm.ThermalNoise.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 receiver thermal noise

Syntax

```
Y = step(H,X)
```

Description `Y = step(H,X)` adds thermal noise to the complex, baseband input signal, `X`, and outputs the result in `Y`. The input signal `X` must be a complex, double or single precision data type column vector or scalar.

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 input signal using parallel concatenated decoding scheme

Description

The `Turbo Decoder` System object decodes the input signal using a parallel concatenated decoding scheme that employs the *a-posteriori* probability (APP) decoder as the constituent decoder. Both constituent decoders use the same trellis structure and algorithm.

Construction

`H = comm.TurboDecoder` creates a System object, `H`. This object uses the *a-posteriori* probability (APP) constituent decoder to iteratively decode the parallel-concatenated convolutionally encoded input data.

`H = comm.TurboDecoder(Name, Value)` creates a turbo decoder object, `H`, with the specified property name set to the specified value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

`H = comm.TurboDecoder(TRELLIS, INTERLVRINDICES, NUMITER)` creates a turbo decoder object, `H`, with the `TrellisStructure` property set to `TRELLIS`, the `InterleaverIndices` property set to `INTERLVRINDICES`, and the `NumIterations` property set to `NUMITER`.

Properties

TrellisStructure

Trellis structure of constituent convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the constituent convolutional code. Use the `istrellis` function to check if a structure is a valid trellis structure. The default is the result of `poly2trellis(4, [13 15], 13)`.

InterleaverIndicesSource

Source of interleaver indices

Specify the source of the interleaver indices as one of `Property | Input port`. When you set this property to `Input port`, the object uses the interleaver indices specified as an input to the step method. When you set this property to `Property`, the object uses the interleaver indices that you specify in the

InterleaverIndices property. When you set this property to Input port, the object processes variable-size signals.

Default: Property

InterleaverIndices

Interleaver indices

Specify the mapping used to permute the input bits at the encoder as a column vector of integers. This mapping is a vector with the number of elements equal to length, L, of the output of the step method. Each element must be an integer between 1 and L, with no repeated values.

Default: (64:-1:1).'.

Algorithm

Decoding algorithm

Specify the decoding algorithm that the object uses for decoding as one of True APP | Max* | Max. When you set this property to True APP, the object implements true *a-posteriori* probability decoding. When you set this property to any other value, the object uses approximations to increase the speed of the computations.

Default: True APP

NumScalingBits

Number of scaling bits

Specify the number of bits the constituent decoders use to scale the input data to avoid losing precision during the computations. The constituent decoders multiply the input by $2^{\text{NumScalingBits}}$ and divide the pre-output by the same factor. The NumScalingBits property must be a scalar integer between 0 and 8. This property applies when you set the Algorithm property to Max*.

Default: 3

NumIterations

Number of decoding iterations

Specify the number of decoding iterations used for each call to the `step` method. The object iterates and provide updates to the log-likelihood ratios (LLR) of the uncoded output bits. The output of the step method is the hard-decision output of the final LLR update.

Default: `6`

Methods

clone	Create Turbo Decoder object with same property values
isLocked	Locked status (logical)
release	Allow property value and input characteristics changes
step	Decode input signal using parallel concatenated decoding scheme

Examples

Transmit turbo-encoded blocks of data over a BPSK-modulated AWGN channel. Then, decode using an iterative turbo decoder and display errors.

```
noiseVar= 4; frmLen = 256;
  s = RandStream('mt19937ar', 'Seed', 11);
  intrlvrIndices = randperm(s, frmLen);

  hTEnc  = comm.TurboEncoder('TrellisStructure', poly2trellis(4, ...
           [13 15 17], 13), 'InterleaverIndices', intrlvrIndices);
  hMod   = comm.BPSKModulator;
  hChan  = comm.AWGNChannel('NoiseMethod', 'Variance', 'Variance', nois
  hTDec  = comm.TurboDecoder('TrellisStructure', poly2trellis(4, ...
           [13 15 17], 13), 'InterleaverIndices', intrlvrIndices, ...
           'NumIterations', 4);
  hError = comm.ErrorRate;
```

```
for frmIdx = 1:8
    data = randi(s, [0 1], frmLen, 1);
    encodedData = step(hTEnc, data);
    modSignal = step(hMod, encodedData);
    receivedSignal = step(hChan, modSignal);

    % Convert received signal to log-likelihood ratios for decodin
    receivedBits  = step(hTDec, (-2/(noiseVar/2))*real(receivedSig

    errorStats = step(hError, data, receivedBits);
end
fprintf('Error rate = %f\nNumber of errors = %d\nTotal bits = %d\n
errorStats(1), errorStats(2), errorStats(3))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Turbo Decoder block reference page. The object properties correspond to the block parameters.

See Also

comm.TurboEncoder | comm.APPDecoder

Purpose

Create Turbo Decoder object with same property values

Syntax

```
C = clone(H)
```

Description

`C = clone(H)` creates a `Turbo Decoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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

Locked status (logical)

Syntax

`TF = isLocked(H)`

Description

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `TurboDecoder` 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.

comm.TurboDecoder.release

Purpose

Allow property value and input characteristics changes

Syntax

```
release(H)
```

Description

`release(H)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 input signal using parallel concatenated decoding scheme

Syntax

```
Y = step(H,X)
Y = step(H, X, INTERLVRINDICES)
```

Description

`Y = step(H,X)` decodes the input data, `X`, using the parallel concatenated convolutional coding scheme that you specify using the `TrellisStructure` and `InterleaverIndices` properties. It returns the binary decoded data, `Y`. Both `X` and `Y` are column vectors of double precision data type. When the constituent convolutional code represents a rate 1/N code, the step method sets the length of the output vector, `Y`, to (`M`-2*numTails)/(2*`N`-1), where `M` represents the input vector length and `numTails` is given by log2(TrellisStructure.numStates)*N. The output length, `L`, is the same as the length of the interleaver indices.

`Y = step(H, X, INTERLVRINDICES)` uses the INTERLVRINDICES specified as an input. INTERLVRINDICES is a column vector containing integer values from `1` to `L` with no repeated values. The lengths of the INTERLVRINDICES input and the `Y` output are the same.

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.

---

comm.TurboEncoder

Purpose

Encode input signal using parallel concatenated encoding scheme

Description

The `Turbo Encoder` System object encodes a binary input signal using a parallel concatenated coding scheme. This coding scheme uses two identical convolutional encoders and appends the termination bits at the end of the encoded data bits.

Construction

`H = comm.TurboEncoder` creates a System object, `H`, that encodes binary data using a turbo encoder.

`H = comm.TurboEncoder(Name, Value)` creates a turbo encoder object, `H`, with the specified property name set to the specified value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1, ,NameN,ValueN`.

`H = comm.TurboEncoder(TRELLIS, INTERLVRINDICES)` creates a turbo encoder object, `H`. In this construction, the `TrellisStructure` property is set to `TRELLIS`, and the `InterleaverIndices` property is set to `INTERLVRINDICES`.

Properties

TrellisStructure

Trellis structure of constituent convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the constituent convolutional code. Use the `istrellis` function to check if a structure is a valid trellis structure. The default is the result of `poly2trellis(4, [13 15], 13)`.

InterleaverIndicesSource

Source of interleaver indices

Specify the source of the interleaver indices as one of `Property | Input port`. When you set this property to `Input port`, the object uses the interleaver indices specified as an input to the step method. When you set this property to `Property`, the object uses the interleaver indices that you specify in the

`InterleaverIndices` property. When you set this property to `Input port`, the object processes variable-size signals.

Default: `Property`

InterleaverIndices

Interleaver indices

Specify the mapping used to permute the input bits at the encoder as a column vector of integers. This mapping is a vector with the number of elements equal to the length of the input for the step method. Each element must be an integer between 1 and L, with no repeated values.

Default: `(64:-1:1).'`.

Methods

clone	Create Turbo Encoder object with same property values
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
step	Encode input signal using parallel concatenated coding scheme

Examples

Transmit turbo-encoded blocks of data over a BPSK-modulated AWGN channel. Then, decode the data using an iterative turbo decoder and display errors.

```
noiseVar= 4; frmLen = 256;
  s = RandStream('mt19937ar', 'Seed', 11);
  intrlvrIndices = randperm(s, frmLen);

  hTEnc  = comm.TurboEncoder('TrellisStructure', poly2trellis(4, ...
           [13 15 17], 13), 'InterleaverIndices', intrlvrIndices);
```

```
hMod   = comm.BPSKModulator;
hChan  = comm.AWGNChannel('NoiseMethod', 'Variance', 'Variance', nois
hTDec  = comm.TurboDecoder('TrellisStructure', poly2trellis(4, ...
         [13 15 17], 13), 'InterleaverIndices', intrlvrIndices, ...
         'NumIterations', 4);
hError = comm.ErrorRate;

for frmIdx = 1:8
    data = randi(s, [0 1], frmLen, 1);
    encodedData = step(hTEnc, data);
    modSignal = step(hMod, encodedData);
    receivedSignal = step(hChan, modSignal);

    % Convert received signal to log-likelihood ratios for decoding
    receivedBits  = step(hTDec, (-2/(noiseVar/2))*real(receivedSignal

    errorStats = step(hError, data, receivedBits);
end
fprintf('Error rate = %f\nNumber of errors = %d\nTotal bits = %d\n',
errorStats(1), errorStats(2), errorStats(3))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Turbo Encoder block reference page. The object properties correspond to the block parameters.

See Also

comm.TurboDecoder | comm.ConvolutionalEncoder

Purpose

Create Turbo Encoder object with same property values

Syntax

`C = clone(H)`

Description

`C = clone(H)` creates a `Turbo Encoder` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

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.

comm.TurboEncoder.isLocked

Purpose Locked status for input attributes and nontunable properties

Syntax `TF = isLocked(H)`

Description Description

`TF = isLocked(H)` returns the locked status, `TF` of the `TurboEncoder` 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)` release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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

Encode input signal using parallel concatenated coding scheme

Syntax

```
Y = step(H,X)
Y = step(H, X, INTERLVRINDICES)
```

Description

`Y = step(H,X)` encodes the input data, `X`, using the parallel concatenated convolutional coding scheme that you specify using the `TrellisStructure` and `InterleaverIndices` properties. It returns the binary decoded data, `Y`. Both `X` and `Y` are column vectors of numeric, logical, or unsigned fixed point with word length 1 (fi object). When the constituent convolutional encoder represents a rate 1/N code, the step method sets the length of the output vector, `Y`, to L*(2*N-1)+2*numTails where `L` represents the input vector length and `numTails` is given by $\log_2$(TrellisStructure.numStates)*`N`. The tail bits, due to the termination, are appended at the end after the input bits are encoded.

`Y = step(H, X, INTERLVRINDICES)` uses the INTERLVRINDICES specified as an input. INTERLVRINDICES is a column vector containing integer values from `1` to `L` with no repeated values. The length of the data input X and the INTERLVRINDICES input must be the same.

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 convolutionally encoded data using Viterbi algorithm

Description

The `ViterbiDecoder` object decodes input symbols to produce binary output symbols. This object can process several symbols at a time for faster performance. This object processes variable-size signals; however, variable-size signals cannot be applied for erasure inputs.

Construction

`H = comm.ViterbiDecoder` creates a Viterbi decoder System object, `H`. This object uses the Viterbi algorithm to decode convolutionally encoded input data.

`H = comm.ViterbiDecoder(Name,Value)` creates a Viterbi decoder 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`).

`H = comm.ViterbiDecoder(TRELLIS,Name,Value)` creates a Viterbi decoder object, `H`. This object has the `TrellisStructure` property set to `TRELLIS` and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the convolutional code. The default is the result of `poly2trellis(7, [171 133])`. Use the `istrellis` function to verify whether a structure is a valid trellis.

InputFormat

Input format

Specify the format of the input to the decoder as `Unquantized | Hard | Soft`. The default is `Unquantized`.

When you set this property to `Unquantized`, the input must be a real vector of double- or single-precision soft values that are unquantized. The object considers negative numbers to be `1`s and positive numbers to be `0`s.

When you set this property to `Hard`, the input must be a vector of hard decision values, which are `0`s or `1`s. The data type of the inputs can be double-precision, single-precision, logical, 8-, 16-, and 32-bit signed integers. You can also use 8-, 16-, and 32-bit unsigned integers.

When you set this property to `Soft`, the input requires a vector of quantized soft values represented as integers between 0 and $2^{SoftInputWordLength} - 1$. The data type of the inputs can be double-precision, single-precision, logical, 8-, 16-, and 32-bit signed integers. You can also use 8-, 16-, and 32-bit unsigned integers. Alternately, you can specify the data type as an unsigned and unscaled fixed point object (fi) with a word length equal to the word length that you specify in the `SoftInputWordLength` property. The object considers negative numbers to be `0`s and positive numbers to be `1`s.

SoftInputWordLength

Soft input word length

Specify the number of bits to represent each quantized soft input value as a positive, integer scalar value. The default is `4` bits. This property applies when you set the `InputFormat` property to `Soft`.

InvalidQuantizedInputAction

Action when input values are out of range

Specify the action the object takes when input values are out of range as `Ignore | Error`. The default is `Ignore`. Set this property to `Error` so that the object generates an error when the quantized input values are out of range. This property applies when you set the `InputFormat` property to `Hard` or `Soft`.

TracebackDepth

Traceback depth

Specify the number of trellis branches to construct each traceback path as a numeric, integer scalar value. The default is `34`. The traceback depth influences the decoding accuracy and delay. The number of zero symbols that precede the first decoded symbol in the output represent a decoding delay.

When you set the `TerminationMethod` property to `Continuous`, the decoding delay consists of `TracebackDepth` zero symbols or `TracebackDepth`$\times K$ zero bits for a rate K/N convolutional code.

When you set the `TerminationMethod` property to `Truncated` or `Terminated`, there is no output delay. In this case, `TracebackDepth` must be less than or equal to the number of symbols in each input. If the code rate is $1/2$, a typical traceback depth value is about five times the constraint length of the code.

TerminationMethod

Termination method of encoded frame

Specify the termination method as `Continuous` | `Truncated` | `Terminated`. The default is `Continuous`.

In `Continuous` mode, the object saves the internal state metric at the end of each frame for use with the next frame. The object treats each traceback path independently.

In `Truncated` mode, the object treats each frame independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state. In `Terminated` mode, the object treats each frame independently, and the traceback path always starts and ends in the all-zeros state.

ResetInputPort

Enable decoder reset input

Set this property to true to enable an additional `step` method input. The default is `false`. When the reset input is a nonzero value, the object resets the internal states of the decoder to initial conditions. This property applies when you set the `TerminationMethod` property to `Continuous`.

DelayedResetAction

Reset on nonzero input via port

Set this property to true to delay resetting the object output. The default is false. When you set this property to true, the reset of the internal states of the decoder occurs after the object computes the decoded data. When you set this property to false, the reset of the internal states of the decoder occurs before the object computes the decoded data. This property applies when you set the `ResetInputPort` property to true.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as `None | Property`. The default is `None`.

When you set this property to `None`, the object assumes no puncturing. Set this property to `Property` to decode punctured codewords based on a puncture pattern vector specified via the `PuncturePattern` property.

PuncturePattern

Puncture pattern vector

Specify puncture pattern to puncture the encoded data. The default is `[1; 1; 0; 1; 0; 1]`. The puncture pattern is a column vector of `1`s and `0`s. The `0`s indicate the position to insert dummy bits. The puncture pattern must match the puncture pattern used by the encoder. This property applies when you set the `PuncturePatternSource` property to `Property`.

ErasuresInputPort

Enable erasures input

Set this property to `true` to specify a vector of erasures as a `step` method input. The default is `false`. The erasures input must be a double-precision or logical, binary, column vector. This vector indicates which symbols of the input codewords to erase. Values of `1` indicate erased bits. The decoder does not update the branch metric for the erasures in the incoming data stream.

The lengths of the `step` method erasure input and the `step` method data input must be the same. When you set this property to `false`, the object assumes no erasures.

OutputDataType

Data type of output

Specify the data type of the output as `Full precision` | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `logical`. The default is `Full precision`.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in `Smallest unsigned integer` or `Full precision` mode.

Fixed-Point Properties

StateMetricDataType

Data type of state metric

Specify the state metric data type as `Full precision` | `Custom`. The default is `Full precision`.

When you set this property to `Full precision`, the object sets the state metric fixed-point type to `numerictype([],16)`. This property applies when you set the `InputFormat` property to `Hard` or `Soft`.

When you set the `InputFormat` property to `Hard`, the `step` method data input must be a column vector. This vector comprises unsigned, fixed point numbers (fi objects) of word length `1` to enable fixed-point Viterbi decoding. Based on this input (either a `0` or a `1`), the object calculates the internal branch metrics using an unsigned integer of word length L. In this case, L indicates the number of output bits as specified by the trellis structure.

When you set the InputFormat property to Soft, the step method data input must be a column vector. This vector comprises unsigned, fixed point numbers (fi objects) of word length N. N indicates the number of soft-decision bits specified in the SoftInputWordLength property.

The step method data inputs must be integers in the range 0 to 2^N–1. The object calculates the internal branch metrics using an unsigned integer of word length $L = (N + Nout - 1)$. In this case, *Nout* represents the number of output bits as specified by the trellis structure.

CustomStateMetricDataType

Fixed-point data type of state metric

Specify the state metric fixed-point type as an unscaled, numerictype object with a signedness of Auto. The default is numerictype([],16). This property applies when you set the StateMetricDataType property to Custom.

Methods

clone	Create Viterbi decoder object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of the Viterbi decoder object
step	Decode convolutionally encoded data using Viterbi algorithm

Examples

Transmit a convolutionally encoded 8-DPSK-modulated bit stream through an AWGN channel. Then, demodulate, decode using a Viterbi decoder, and count errors.

```
hConEnc = comm.ConvolutionalEncoder;
hMod = comm.DPSKModulator('BitInput',true);
hChan = comm.AWGNChannel('NoiseMethod', ...
                   'Signal to noise ratio (SNR)', 'SNR',10);
hDemod = comm.DPSKDemodulator('BitOutput',true);
hDec = comm.ViterbiDecoder('InputFormat','Hard');
% Delay in bits is TracebackDepth times the number of bits per symbol
  delay = hDec.TracebackDepth*...
                   log2(hDec.TrellisStructure.numInputSymbols);
  hError = comm.ErrorRate('ComputationDelay',3,'ReceiveDelay',delay);
  for counter = 1:20
     data = randi([0 1],30,1);
     encodedData = step(hConEnc, data);
     modSignal = step(hMod, encodedData);
     receivedSignal = step(hChan, modSignal);
     demodSignal = step(hDemod, receivedSignal);
     receivedBits = step(hDec, demodSignal);
     errorStats = step(hError, data, receivedBits);
   end
   fprintf('Error rate = %f\nNumber of errors = %d\n', ...
     errorStats(1), errorStats(2))
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Viterbi Decoder block reference page. The object properties correspond to the block parameters, except:

- The **Decision type** parameter corresponds to the `InputFormat` property.
- The **Operation mode** parameter corresponds to the `TerminationMethod` property.

See Also

`comm.ConvolutionalEncoder` | `comm.APPDecoder`

comm.ViterbiDecoder.clone

Purpose Create Viterbi decoder object with same property values

Syntax `C = clone(H)`

Description `C = clone(H)` creates a `ViterbiDecoder` 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 a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(H)`

comm.ViterbiDecoder.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description `TF = isLocked(H)` returns the locked status, `TF` of the `ViterbiDecoder` 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.

comm.ViterbiDecoder.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)`Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

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 the Viterbi decoder object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `ViterbiDecoder` object, `H`.

comm.ViterbiDecoder.step

Purpose Decode convolutionally encoded data using Viterbi algorithm

Syntax

```
Y = step(H,X)
Y = step(H,X,ERASURES)
Y = step(H,X,R)
```

Description `Y = step(H,X)` decodes encoded data, `X`, using the Viterbi algorithm and returns `Y`. `X`, must be a column vector with data type and values that depend on how you set the `InputFormat` property. If the convolutional code uses an alphabet of 2^N possible symbols, the length of the input vector, `X`, must be $L \times N$ for some positive integer L. Similarly, if the decoded data uses an alphabet of 2^K possible output symbols, the length of the output vector, `Y`, is $L \times K$.

`Y = step(H,X,ERASURES)` uses the binary column input vector, `ERASURES`, to erase the symbols of the input codewords. The elements in `ERASURES` must be of data type double or logical. Values of `1` in the `ERASURES` vector correspond to erased symbols, and values of `0` correspond to non-erased symbols. The lengths of the `X` and `ERASURES` inputs must be the same. This syntax applies when you set the `ErasuresInputPort` property to true.

`Y = step(H,X,R)` resets the internal states of the decoder when you input a non-zero reset signal, `R`. `R` must be a double precision or logical scalar. This syntax applies when you set the `TerminationMethod` property to `Continuous` and 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.

Purpose

Generate Walsh code from orthogonal set of codes

Description

The `WalshCode` object generates a Walsh code from an orthogonal set of codes.

Construction

`H = comm.WalshCode` creates a Walsh code generator System object, `H`. This object generates a Walsh code from a set of orthogonal codes.

`H = comm.WalshCode(Name,Value)` creates a Walsh code generator 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

Length

Length of generated code

Specify the length of the generated code as a numeric, integer scalar value that is a power of two. The default is `64`.

Index

Index of code of interest

Specify the index of the desired code from the available set of codes as a numeric, integer scalar value in the range `[0, 1, ... , N-1]`. N is the value of the `Length` property. The default is `60`. The number of zero crossings in the generated code equals the value of the specified index.

SamplesPerFrame

Number of output samples per frame

Specify the number of Walsh code samples that the step method outputs as a numeric, positive, integer scalar value . The default is `1`. If you set this property to a value of M, then the `step` method outputs M samples of a Walsh code of length N. N is the length of the code that you specify in the `Length` property.

OutputDataType

Data type of output

Specify the output data type as `double | int8`. The default is `double`.

Methods

clone	Create Walsh code generator object with same property values
getNumInputs	Number of expected inputs to step method
getNumOutputs	Number of outputs from step method
isLocked	Locked status for input attributes and nontunable properties
release	Allow property value and input characteristics changes
reset	Reset states of Walsh code generator object
step	Generate Walsh code from orthogonal set of codes

Examples

Generate 10 samples of a length-64 Walsh code sequence.

```
hwc = comm.WalshCode('SamplesPerFrame', 10);
seq = step(hwc)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Walsh Code Generator block reference page. The object properties correspond to the block parameters, except:

- The object does not have a property to select frame based outputs.
- The object does not have a property that corresponds to the **Sample time** parameter.

See Also

`comm.HadamardCode` | `comm.OVSFCode`

comm.WalshCode.clone

Purpose Create Walsh code generator object with same property values

Syntax

```
C = clone(H)
```

Description `C = clone(H)` creates a `WalshCode` object `C`, with the same property values as `H`. The `clone` method creates a new unlocked object with uninitialized states.

comm.WalshCode.getNumInputs

Purpose

Number of expected inputs to step method

Syntax

`N = getNumInputs(H)`

Description

`N = getNumInputs(H)` returns a positive integer, `N`, representing the number of expected inputs to the `step` method. This value will change if any properties that turn inputs on or off are changed. The `step` method must be called with a number of input arguments equal to the result of `getNumInputs(`*`H`*`)`

comm.WalshCode.getNumOutputs

Purpose Number of outputs from step method

Syntax `N = getNumOutputs(H)`

Description `N = getNumOutputs(H)` returns the number of outputs, `N`, from the `step` method. This value will change if any properties that turn inputs on or off are changed.

Purpose Locked status for input attributes and nontunable properties

Syntax

```
TF = isLocked(H)
```

Description

`TF = isLocked(H)` returns the locked status, `TF` of the `WalshCode` 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.

comm.WalshCode.release

Purpose Allow property value and input characteristics changes

Syntax `release(H)`

Description `release(H)` Release system resources (such as memory, file handles or hardware connections) and allows all properties and input characteristics to be changed.

Note You cannot use the `release` method on System objects in code generated from MATLAB.

Purpose Reset states of Walsh code generator object

Syntax `reset(H)`

Description `reset(H)` resets the states of the `WalshCode` object, `H`.

comm.WalshCode.step

Purpose Generate Walsh code from orthogonal set of codes

Syntax `Y = step(H)`

Description `Y = step(H)` outputs a frame of the Walsh code in column vector `Y`. Specify the frame length with the `SamplesPerFrame` property. The Walsh code corresponds to a row of an NxN Hadamard matrix, where N is a nonnegative power of 2 that you specify in the `Length` property. Use the `Index` property to choose the row of the Hadamard matrix. The output code is in a bi-polar format with `0` and `1` mapped to `1` and `-1` 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

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

cloneImpl	Copy System object
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
processInputSizeChangeImpl	Action when input size changes
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.StringSet | matlab.system.StringSet | matlab.system.StringSet | matlab.system.mixin.FiniteSource | matlab.system.mixin.FiniteSource | matlab.system.mixin.FiniteSource | matlab.system.mixin.FiniteSource`

How To

- “Object-Oriented Programming”
- Class Attributes
- Property Attributes
- “Method Attributes”
- “Define Basic System Objects”
- “Define Basic System Objects”
- “Define Basic System Objects”
- “Define Basic System Objects”
- “Define Property Attributes”
- “Define Property Attributes”
- “Define Property Attributes”
- “Define Property Attributes”

Purpose

Copy System object

Syntax

`cloneImpl(obj)`

Description

`cloneImpl(obj)` copies a System object by using the `saveObjectImpl` and `loadObjectImpl` methods. The default `cloneImpl` copies an object and its current state but does not copy any private or protected properties. If the object you clone is locked and you use the default `cloneImpl`, the new object will also be locked. If you define your own `cloneImpl` and the associated `saveObjectImpl` and `loadObjectImpl`, you can specify whether to clone the object's state and whether to clone the object's private and protected properties.

`cloneImpl` is called by the `clone` method.

Note You must set `Access=protected` for this method.

Input Arguments

obj

System object handle of object to clone.

Examples

Use the `cloneImpl` method to copy a System object

```
methods (Access=protected)
   function obj2 = cloneImpl(obj1)
      s = saveObject (obj1);
      obj2 = loadObject(s);
   end
end
```

See Also

`saveObjectImpl` | `saveObjectImpl` | `saveObjectImpl` | `saveObjectImpl` | `loadObjectImpl` | `loadObjectImpl` | `loaObjectImpl` | `saveObjectImpl`

How To

- "Clone System Object"

- “Clone System Object”
- “Clone System Object”
- “Clone System Object”

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.

`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` | `setupImpl` | `setupImpl` | `setupImpl`

How To

- "Define Property Attributes"
- "Define Property Attributes"
- "Define Property Attributes"
- "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.

Do not set any object properties in this `getNumInputsImpl` 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` | `setupImpl` | `setupImpl` | `setupImpl` | `stepImpl` | `stepImpl` | `stepImpl` | `stepImpl` | `getNumOutputsImpl` | `getNumOutputsImpl` | `getNumOutputsImpl` | `getNumOutputsImpl`

How To

- “Change Number of Step Inputs or Outputs”
- “Change Number of Step Inputs or Outputs”
- “Change Number of Step Inputs or Outputs”
- “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.

Do not set any object properties in this `getNumOutputsImpl` 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 = 0;
   end
end
```

See Also

`stepImpl` | `stepImpl` | `stepImpl` | `stepImpl` | `getNumInputsImpl` | `getNumInputsImpl` | `getNumInputsImpl` | `getNumInputsImpl` | `setupImpl` | `setupImpl` | `setupImpl` | `setupImpl`

How To

- "Change Number of Step Inputs or Outputs"
- "Change Number of Step Inputs or Outputs"
- "Change Number of Step Inputs or Outputs"
- "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 public, non-state 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

Public, non-state 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` | `setProperties` | `setProperties` | `setProperties`

How To

- “Hide Inactive Properties”
- “Hide Inactive Properties”
- “Hide Inactive Properties”
- “Hide Inactive Properties”

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 Arguments

obj

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` | `validatePropertiesImpl` | `validatePropertiesImpl` | `validatePropertiesImpl` | `validatePropertiesImpl` | `setProperties` | `setProperties` | `setProperties` | `setProperties`

How To

- "Validate Property and Input Values"
- "Validate Property and Input Values"
- "Validate Property and Input Values"
- "Validate Property and Input Values"
- "Define Property Attributes"
- "Define Property Attributes"
- "Define Property Attributes"
- "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` | `resetImpl` | `resetImpl` | `resetImpl`

How To

- “Release System Object Resources”
- “Release System Object Resources”
- “Release System Object Resources”
- “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` | `releaseImpl` | `releaseImpl` | `releaseImpl`

How To

- “Reset Algorithm State”
- “Reset Algorithm State”
- “Reset Algorithm State”
- “Reset Algorithm State”

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 objectSystem 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"
- "Set Property Values at Construction Time"
- "Set Property Values at Construction Time"
- "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 < 0
```

```
            error('Opening the file failed');
        end
    end
end
```

See Also

validatePropertiesImpl | validatePropertiesImpl | validatePropertiesImpl | validatePropertiesImpl | validateInputsImpl | validateInputsImpl | validateInputsImpl | validateInputsImpl | setProperties | setProperties | setProperties | setProperties

How To

- “Initialize Properties and Setup One-Time Calculations”
- “Initialize Properties and Setup One-Time Calculations”
- “Initialize Properties and Setup One-Time Calculations”
- “Initialize Properties and Setup One-Time Calculations”
- “Set Property Values at Construction Time”
- “Set Property Values at Construction Time”
- “Set Property Values at Construction Time”
- “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` | `getNumInputsImpl` | `getNumInputsImpl` | `getNumInputsImpl` | `getNumOutputsImpl` | `getNumOutputsImpl` | `getNumOutputsImpl` | `getNumOutputsImpl` | `validateInputsImpl` | `validateInputsImpl` | `validateInputsImpl` | `validateInputsImpl`

How To

- "Define Basic System Objects"
- "Define Basic System Objects"
- "Define Basic System Objects"
- "Define Basic System Objects"
- "Change Number of Step Inputs or Outputs"
- "Change Number of Step Inputs or Outputs"
- "Change Number of Step Inputs or Outputs"
- "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` | `validatePropertiesImpl` | `validatePropertiesImpl` | `validatePropertiesImpl` | `setupImpl` | `setupImpl` | `setupImpl` | `setupImpl`

How To

- “Validate Property and Input Values”
- “Validate Property and Input Values”
- “Validate Property and Input Values”
- “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` | `processTunedPropertiesImpl` | `processTunedPropertiesImpl` | `processTunedPropertiesImpl` | `setupImpl` | `setupImpl` | `setupImpl` | `setupImpl` | `validateInputsImpl` | `validateInputsImpl` | `validateInputsImpl` | `validateInputsImpl`

How To

- “Validate Property and Input Values”
- “Validate Property and Input Values”
- “Validate Property and Input Values”
- “Validate Property and Input Values”

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”
- “Load System Object”
- “Load System Object”
- “Load System Object”
- “Save System Object”
- “Save System Object”
- “Save System Object”
- “Save System Object”

matlab.System.processInputSizeChangeImpl

Purpose

Action when input size changes

Syntax

`processInputSizeChangeImpl(obj,input1,...,inputN)`

Description

`processInputSizeChangeImpl(obj,input1,...,inputN)` specifies the actions to perform when any `step` method input changes size (after the first call to `step`).

`processInputSizeChangeImpl` is called by the `step` method before the `stepImpl` method is called.

Note You must set `Access=protected` for this method.

Tips

Use this method when property values or settings depend on the size of the input. For example, you may want to reset some or all of the states in the object when the input sizes change.

Input Arguments

obj

System object handle

input1,...,inputN

Inputs to the System object `step` method.

Examples

Use `processInputSizeChangeImpl` to have the object reset if the input size changes. In this case, `ResetOnSizeChange` is a property of the object. If `ResetOnSizeChange` is true, then reset is called what an input size changes.

```
methods (Access=protected)
   function processInputSizeChangeImpl(obj, ~)
      if obj.ResetOnSizeChange
        reset(obj);
      end
    end
```

```
end
```

See Also

resetImpl | resetImpl | resetImpl | resetImpl | stepImpl | stepImpl | stepImpl | stepImpl

How To

- "Process Input Size Change"
- "Process Input Size Change"
- "Process Input Size Change"
- "Process Input Size Change"

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 properties. 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”
- “Save System Object”
- “Save System Object”
- “Save System Object”
- “Load System Object”
- “Load System Object”
- “Load System Object”
- “Load System Object”

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` | `matlab.System` | `matlab.System` | `matlab.System`

Tutorials

- "Define Finite Source Objects"
- "Define Finite Source Objects"
- "Define Finite Source Objects"
- "Define Finite Source Objects"

How To

- "Object-Oriented Programming"
- Class Attributes
- Property Attributes

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` |
`matlab.system.mixin.FiniteSource` |
`matlab.system.mixin.FiniteSource` |
`matlab.system.mixin.FiniteSource`

How To

- “Define Finite Source Objects”
- “Define Finite Source Objects”

- “Define Finite Source Objects”
- “Define Finite Source Objects”

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` | `matlab.System` | `matlab.System` | `matlab.System`

How To

- "Object-Oriented Programming"
- Class Attributes
- Property Attributes
- "Limit Property Values to Finite String Set"
- "Limit Property Values to Finite String Set"
- "Limit Property Values to Finite String Set"
- "Limit Property Values to Finite String Set"

Index

A

A-Law Compressor block 2-2
A-Law Expander block 2-4
AGC 2-21
`algdeintrlv` function 1-2
Algebraic Deinterleaver block 2-6
Algebraic Interleaver block 2-9
`algintrlv` function 1-4
Align Signals block 2-13
`alignsignals` function 1-7
`amdemod` function 1-15
`ammod` function 1-17
APP Decoder block 2-17
`arithdeco` function 1-19
`arithenco` function 1-20
asymmetrical Jakes Doppler spectrum
    object 1-200
AWGN Channel block 2-28
`awgn` function 1-22

B

Barker Code Generator block 2-35
Baseband PLL block 2-37
BCH coding
    sample code
        using various coding methods 1-237
BCH Decoder block 2-40
BCH Encoder block 2-47
`bchdec` function 1-24
`bchenc` function 1-29
`bchgenpoly` function 1-36
`bchnumerr` function 1-38
`berawgn` function 1-39
`bercoding` function 1-43
`berconfint` function 1-47
`berfading` function 1-49
`berfit` function 1-54
Bernoulli Binary Generator block 2-58
`bersync` function 1-63
`bertool` function 1-66
bi-Gaussian Doppler spectrum object 1-206
`bi2de` function 1-67
`bin2gray` function 1-69
Binary Cyclic Decoder block 2-61
Binary Cyclic Encoder block 2-64
Binary Linear Decoder block 2-74
Binary Linear Encoder block 2-77
binary matrix format
    sample code 1-236
Binary Symmetric Channel block 2-86
binary vector format
    sample code 1-236
Binary-Input RS Encoder block 2-67
Binary-Output RS Decoder block 2-79
binary-to-decimal conversion 1-67
Bipolar to Unipolar Converter block 2-89
Bit to Integer Converter block 2-93
`biterr` function 1-72
Bose-Chaudhuri-Hocquenghem (BCH) coding
    sample code
        using various coding methods 1-237
BPSK Demodulator Baseband block 2-97
BPSK Modulator Baseband block 2-106

C

channel visualization tool
    `plot (channel)` 1-413
Charge Pump PLL block 2-110
CMA Equalizer block 2-114
`cma` function 1-82
`comm_links` function 1-84
`commlib` function 1-85
`commscope` package 1-86
`commscope.eyediagram` 1-87
`commstartup` function 1-137
`compand` function 1-150
Complex Phase Difference block 2-118
Complex Phase Shift block 2-119

conjugate elements in Galois fields
 even number of field elements 1-162
 odd number of field elements 1-286
constellations
 hexagonal
  sample code 1-276
convdeintrlv function 1-153
convenc function 1-155
conversion
 binary to decimal 1-67
 decimal to binary 1-179
 octal to decimal 1-408
 vectors to matrices 1-528
convintrlv function 1-158
convmtx function 1-160
Convolutional Deinterleaver block 2-123
Convolutional Encoder block 2-126
Convolutional Interleaver block 2-132
cosets
 even number of field elements 1-162
 odd number of field elements 1-286
cosets function 1-162
CPFSK Demodulator Baseband block 2-135
CPFSK Modulator Baseband block 2-140
CPM Demodulator Baseband block 2-145
CPM Modulator Baseband block 2-153
CPM Phase Recovery block 2-161
CRC-N Generator block 2-165
CRC-N Syndrome Detector block 2-169
cyclic coding
 sample code 1-237
  for tracking errors 1-184
  using various coding methods 1-237
cyclotomic cosets
 even number of field elements 1-162
 odd number of field elements 1-286
cyclpoly function 1-176

D

Data Mapper block 2-172
DBPSK Demodulator Baseband block 2-175
DBPSK Modulator Baseband block 2-178
de2bi function 1-179
decimal format
 sample code 1-236
decode function 1-182
Deinterlacer block 2-181
deintrlv function 1-186
Derepeat block 2-183
Descrambler block 2-188
dfe function 1-187
dftmtx function 1-193
Differential Decoder block 2-190
Differential Encoder block 2-193
Discrete-Time Eye Diagram Scope block 2-196
Discrete-Time Scatter Plot Scope block 2-207
Discrete-Time Signal Trajectory Scope
 block 2-217
Discrete-Time VCO block 2-225
distspec function 1-195
doppler package 1-199
doppler.ajakes object 1-200
doppler.bigaussian object 1-206
doppler.flat object 1-210
doppler.gaussian object 1-212
doppler.jakes object 1-215
doppler.rjakes object 1-217
doppler.rounded object 1-221
dpcmdeco function 1-224
dpcmenco function 1-225
dpcmopt function 1-227
dpskdemod function 1-229
dpskmod function 1-231
DQPSK Demodulator Baseband block 2-228
DQPSK Modulator Baseband block 2-231
DSB AM Demodulator Passband block 2-236
DSB AM Modulator Passband block 2-239
DSBSC AM Demodulator Passband block 2-241

DSBSC AM Modulator Passband block 2-244
dvbs2ldpc function 1-233

E

Early-Late Gate Timing Recovery block 2-246
encode function 1-234
equalize function 1-239
Error Rate Calculation block 2-251
eyediagram function 1-241

F

fft function 1-259
filter function
  as a channel 1-260
  Galois fields 1-261
filters
  Galois fields
    odd number of field elements 1-293
Find Delay block 2-269
finddelay function 1-262
flat Doppler spectrum object 1-210
FM Demodulator Passband block 2-274
FM Modulator Passband block 2-277
fmdemod function 1-267
fmmod function 1-268
Free Space Path Loss block 2-279
fskdemod function 1-269
fskmod function 1-271

G

Gardner Timing Recovery block 2-282
Gaussian Doppler spectrum object 1-212
Gaussian Filter block 2-288
Gaussian Noise Generator block 2-298
gen2par function 1-273
General Block Deinterleaver block 2-302
General Block Interleaver block 2-304
General CRC Detector HDL Optimized
  block 2-324
General CRC Generator block 2-306
General CRC Generator HDL Optimized
  block 2-313
General CRC Syndrome Detector block 2-319
General Multiplexed Deinterleaver block 2-329
General Multiplexed Interleaver block 2-331
General QAM Demodulator Baseband
  block 2-334
General QAM Modulator Baseband block 2-349
General TCM Decoder block 2-353
General TCM Encoder block 2-358
genqamdemod function 1-275
genqammod function 1-276
gf function 1-278
gfadd function 1-281
gfconv function 1-283
gfcosets function 1-286
gfdeconv function 1-288
gfdiv function 1-291
gffilter function 1-293
gflineq function 1-295
gfminpol function 1-297
gfmul function 1-299
gfpretty function 1-301
gfprimck function 1-303
gfprimdf function 1-305
gfprimfd function 1-307
gfrank function 1-310
gfrepcov function 1-311
gfroots function 1-313
gfsub function 1-315
gftable function 1-317
gftrunc function 1-319
gftuple function 1-320
gfweight function 1-323
GMSK Demodulator Baseband block 2-364
GMSK Modulator Baseband block 2-369
Gold Sequence Generator block 2-374

gray2bin function 1-325

H

Hadamard Code Generator block 2-383
hammgen function 1-327
Hamming coding
sample code
using various coding methods 1-237
using various formats 1-236
Hamming Decoder block 2-386
Hamming Encoder block 2-389
Hamming weight 1-323
hank2sys function 1-330
heldeintrlv function 1-332
Helical Deinterleaver block 2-392
Helical Interleaver block 2-396
helintrlv function 1-335
helscandeintrlv function 1-339
helscanintrlv function 1-341
hilbiir function 1-343
huffmandeco function 1-347
huffmandict function 1-349
huffmanenco function 1-352

I

I/Q Imbalance block 2-456
Ideal Rectangular Pulse Filter block 2-400
ifft function 1-353
Insert Zero block 2-410
intdump function 1-354
Integer to Bit Converter block 2-438
Integer-Input RS Decoder HDL Optimized block 2-432
Integer-Input RS Encoder block 2-414
Integer-Input RS Encoder HDL Optimized block 2-420
Integer-Output RS Decoder block 2-426
Integrate and Dump block 2-445
Interlacer block 2-454
intrlv function 1-355
inverses in Galois fields
odd number of elements 1-291
iscatastrophic function 1-357
isprimitive function 1-358
istrellis function 1-360

J

Jakes Doppler spectrum object 1-215

K

Kasami Sequence Generator block 2-464

L

LDPC Decoder block 2-472
LDPC Encoder block 2-478
lineareq function 1-364
Linearized Baseband PLL block 2-481
lloyds function 1-368
LLR algorithm 2-472
LMS Decision Feedback Equalizer block 2-484
lms function 1-371
LMS Linear Equalizer block 2-489
log function 1-373
log-likelihood ratio (LLR) 2-472

M

M-DPSK Demodulator Baseband block 2-504
M-DPSK Modulator Baseband block 2-507
M-FSK Demodulator Baseband block 2-527
M-FSK Modulator Baseband block 2-534
M-PAM Demodulator Baseband block 2-555
M-PAM Modulator Baseband block 2-565
M-PSK Demodulator Baseband block 2-572
M-PSK Modulator Baseband block 2-587
M-PSK Phase Recovery block 2-595

M-PSK TCM Decoder block 2-599
M-PSK TCM Encoder block 2-603
`marcumq` function 1-377
`mask2shift` function 1-379
`matdeintrlv` function 1-381
`matintrlv` function 1-383
Matrix Deinterleaver block 2-494
Matrix Helical Scan Deinterleaver block 2-496
Matrix Helical Scan Interleaver block 2-499
Matrix Interleaver block 2-502
Memoryless Nonlinearity block 2-512
minimum distance 1-323
`minpol` function 1-384
`mldivide` function 1-386
MLSE Equalizer block 2-541
`mlseeq` function 1-388
`modnorm` function 1-393
MSK Demodulator Baseband block 2-608
MSK Modulator Baseband block 2-613
MSK-Type Signal Timing Recovery block 2-617
`mskdemod` function 1-395
`mskmod` function 1-398
Mu-Law Compressor block 2-627
Mu-Law Expander block 2-629
Mueller-Muller Timing Recovery block 2-622
Multipath Rayleigh Fading Channel block 2-631
Multipath Rician Fading Channel block 2-641
multiple roots over Galois fields
    odd number of field elements 1-313
`muxdeintrlv` function 1-400
`muxintrlv` function 1-402

N

`noisebw` function 1-404
Normalized LMS Decision Feedback Equalizer
    block 2-647
Normalized LMS Linear Equalizer block 2-652
`normlms` function 1-406

O

`oct2dec` function 1-408
octal
    conversion to decimal 1-408
OQPSK Demodulator Baseband block 2-657
OQPSK Modulator Baseband block 2-668
`oqpskdemod` function 1-409
`oqpskmod` function 1-410
OVSF Code Generator block 2-694

P

`pamdemod` function 1-411
`pammod` function 1-412
Phase Noise block 2-710
Phase/Frequency Offset block 2-700
Phase-Locked Loop block 2-707
`plot (channel)` function 1-413
PM Demodulator Passband block 2-715
PM Modulator Passband block 2-718
`pmdemod` function 1-414
`pmmod` function 1-415
PN Sequence Generator block 2-720
Poisson Integer Generator block 2-733
`poly2trellis` function 1-416
`primpoly` function 1-420
`pskdemod` function 1-423
`pskmod` function 1-426
Puncture block 2-736

Q

`qamdemod` function 1-427
`qammod` function 1-429
`qfunc` function 1-430
`qfuncinv` function 1-431
QPSK Demodulator Baseband block 2-739
QPSK Modulator Baseband block 2-748
`quantiz` function 1-433
Quantizing Decoder block 2-753

Quantizing Encoder block 2-756

R

Raised Cosine Receive Filter block 2-759
Raised Cosine Transmit Filter block 2-770
`randdeintrlv` function 1-435
`randerr` function 1-436
`randint` function 1-439
`randintrlv` function 1-441
Random Deinterleaver block 2-780
Random Integer Generator block 2-782
Random Interleaver block 2-785
`randseed` function 1-442
`randsrc` function 1-444
rank
    in Galois fields
        odd number of elements 1-310
Rayleigh Noise Generator block 2-787
`rayleighchan` function 1-446
`rcosfir` function 1-453
`rcosflt` function 1-456
`rcosiir` function 1-460
`rcosine` function 1-463
Receiver Thermal Noise block 2-790
Rectangular QAM Demodulator Baseband block 2-793
Rectangular QAM Modulator Baseband block 2-806
Rectangular QAM TCM Decoder block 2-813
Rectangular QAM TCM Encoder block 2-817
`rectpulse` function 1-465
Repeat block 2-824
`reset` function
    for channels 1-467
    for equalizers 1-469
restricted Jakes Doppler spectrum object 1-217
Rician Noise Generator block 2-825
`ricianchan` function 1-470
RLS Decision Feedback Equalizer block 2-829
`rls` function 1-477
RLS Linear Equalizer block 2-834
rounded Doppler spectrum object 1-221
`rsdec` function 1-480
`rsdecof` function 1-483
`rsenc` function 1-484
`rsencof` function 1-486
`rsgenpoly` function 1-488
`rsgenpolycoeffs` function 1-491

S

`scatterplot` function 1-492
Scrambler block 2-839
`semianalytic` function 1-494
`shift2mask` function 1-498
Sign LMS Decision Feedback Equalizer block 2-841
Sign LMS Linear Equalizer block 2-846
signal constellations
    hexagonal
        sample code 1-276
`signlms` function 1-501
Squaring Timing Recovery block 2-851
SSB AM Demodulator Passband block 2-854
SSB AM Modulator Passband block 2-857
`ssbdemod` function 1-504
`ssbmod` function 1-506
`stdchan` function 1-507
`symerr` function 1-515
`syndtable` function 1-519

T

Turbo Decoder 2-862
Turbo Encoder 2-867

U

Uniform Noise Generator block 2-873
Unipolar to Bipolar Converter block 2-876

V

Variable Step LMS Decision Feedback Equalizer
block 2-880
Variable Step LMS Linear Equalizer block 2-885
`varlms` function 1-525
`vec2mat` function 1-528
`vitdec` function 1-530
Viterbi Decoder block 2-890
Voltage-Controlled Oscillator block 2-120

W

Walsh Code Generator block 2-909
weight, Hamming 1-323
`wgn` function 1-538
Windowed Integrator block 2-912